Javascript Algorithms And Data Structures Notes

Patrick Adams

November 1, 2020

Note: This is a draft copy of notes generated by free code camp.
https://www.freecodecamp.org/

Contents

1 Basic Javascript

1.1 Comment Your JavaScript Code

Comments are lines of code that JavaScript will intentionally ignore. Comments are a great way to leave notes to yourself and to other people who will later need to figure out what that code does.
There are two ways to write comments in JavaScript:
Using // will tell JavaScript to ignore the remainder of the text on the current line:
“‘js
// This is an in-line comment.
“‘
You can make a multi-line comment beginning with /* and ending with */:
“‘js
/* This is a
multi-line comment */
“‘
Best PracticeAs you write code, you should regularly add comments to clarify the function of parts of your code. Good commenting can help communicate the intent of your code—both for others and for your future self.

1.2 Declare JavaScript Variables

In computer science, data is anything that is meaningful to the computer. JavaScript provides eight different data types which are undefined, null, boolean, string, symbol, bigint, number, and object.
For example, computers distinguish between numbers, such as the number 12, and strings, such as "12", "dog", or "123 cats", which are collections of characters. Computers can perform mathematical operations on a number, but not on a string.
Variables allow computers to store and manipulate data in a dynamic fashion. They do this by using a "label" to point to the data rather than using the data itself. Any of the eight data types may be stored in a variable.
Variables are similar to the x and y variables you use in mathematics, which means they’re a simple name to represent the data we want to refer to. Computer variables differ from mathematical variables in that they can store different values at different times.
We tell JavaScript to create or declare a variable by putting the keyword var in front of it, like so:
“‘js
var ourName;
“‘
creates a variable called ourName. In JavaScript we end statements with semicolons.
Variable names can be made up of numbers, letters, and $ or _, but may not contain spaces or start with a number.

1.3 Storing Values with the Assignment Operator

In JavaScript, you can store a value in a variable with the assignment operator (=).
myVariable = 5;
This assigns the Number value 5 to myVariable.
If there are any calculations to the right of the = operator, those are performed before the value is assigned to the variable on the left of the operator.
“‘js
var myVar;
myVar = 5;
“‘
First, this code creates a variable named myVar. Then, the code assigns 5 to myVar. Now, if myVar appears

again in the code, the program will treat it as if it is 5.

1.4 Assigning the Value of One Variable to Another

After a value is assigned to a variable using the assignment operator, you can assign the value of that variable to another variable using the assignment operator.
“‘js
var myVar;
myVar = 5;
var myNum;
myNum = myVar;
“‘
The above declares a ‘myVar‘ variable with no value, then assigns it the value ‘5‘. Next, a variable named ‘myNum‘ is declared with no value. Then, the contents of ‘myVar‘ (which is ‘5‘) is assigned to the variable ‘myNum‘. Now, ‘myNum‘ also has the value of ‘5‘.

1.5 Initializing Variables with the Assignment Operator

It is common to initialize a variable to an initial value in the same line as it is declared.
var myVar = 0;
Creates a new variable called myVar and assigns it an initial value of 0.

1.6 Understanding Uninitialized Variables

When JavaScript variables are declared, they have an initial value of undefined. If you do a mathematical operation on an undefined variable your result will be NaN which means "Not a Number". If you concatenate a string with an undefined variable, you will get a literal string of "undefined".

1.7 Understanding Case Sensitivity in Variables

In JavaScript all variables and function names are case sensitive. This means that capitalization matters.
MYVAR is not the same as MyVar nor myvar. It is possible to have multiple distinct variables with the same name but different casing. It is strongly recommended that for the sake of clarity, you do not use this language feature.
Best Practice
Write variable names in JavaScript in camelCase. In camelCase, multi-word variable names have the first word in lowercase and the first letter of each subsequent word is capitalized.
Examples:
“‘js
var someVariable;
var anotherVariableName;
var thisVariableNameIsSoLong;
“‘

1.8 Add Two Numbers with JavaScript

Number is a data type in JavaScript which represents numeric data.
Now let’s try to add two numbers using JavaScript.
JavaScript uses the + symbol as an addition operator when placed between two numbers.
Example:

“‘js
myVar = 5 + 10; // assigned 15
“‘

1.9 Subtract One Number from Another with JavaScript

We can also subtract one number from another.
JavaScript uses the - symbol for subtraction.
Example
“‘js
myVar = 12 - 6; // assigned 6
“‘

1.10 Multiply Two Numbers with JavaScript

We can also multiply one number by another.
JavaScript uses the * symbol for multiplication of two numbers.
Example
“‘js
myVar = 13 * 13; // assigned 169
“‘

1.11 Divide One Number by Another with JavaScript

We can also divide one number by another.
JavaScript uses the / symbol for division.
Example
“‘js
myVar = 16 / 2; // assigned 8
“‘

1.12 Increment a Number with JavaScript

You can easily increment or add one to a variable with the ++ operator.
i++;
is the equivalent of
i = i + 1;
NoteThe entire line becomes i++;, eliminating the need for the equal sign.

1.13 Decrement a Number with JavaScript

You can easily decrement or decrease a variable by one with the -- operator.
i--;
is the equivalent of
i = i - 1;
NoteThe entire line becomes i--;, eliminating the need for the equal sign.

1.14 Create Decimal Numbers with JavaScript

We can store decimal numbers in variables too. Decimal numbers are sometimes referred to as floating point numbers or floats.
NoteNot all real numbers can accurately be represented in floating point. This can lead to rounding errors. Details Here.

1.15 Multiply Two Decimals with JavaScript

In JavaScript, you can also perform calculations with decimal numbers, just like whole numbers.
Let's multiply two decimals together to get their product.

1.16 Divide One Decimal by Another with JavaScript

Now let's divide one decimal by another.

1.17 Finding a Remainder in JavaScript

The remainder operator % gives the remainder of the division of two numbers.
Example
5 % 2 = 1 becauseMath.floor(5 / 2) = 2 (Quotient)2 * 2 = 45 - 4 = 1 (Remainder)
UsageIn mathematics, a number can be checked to be even or odd by checking the remainder of the division of the number by 2.
17 % 2 = 1 (17 is Odd)48 % 2 = 0 (48 is Even)
NoteThe remainder operator is sometimes incorrectly referred to as the "modulus" operator. It is very similar to modulus, but does not work properly with negative numbers.

1.18 Compound Assignment With Augmented Addition

In programming, it is common to use assignments to modify the contents of a variable. Remember that everything to the right of the equals sign is evaluated first, so we can say:
myVar = myVar + 5;
to add 5 to myVar. Since this is such a common pattern, there are operators which do both a mathematical operation and assignment in one step.
One such operator is the += operator.
“‘js
var myVar = 1;
myVar += 5;
console.log(myVar); // Returns 6
“‘

1.19 Compound Assignment With Augmented Subtraction

Like the += operator, -= subtracts a number from a variable.
myVar = myVar - 5;
will subtract 5 from myVar. This can be rewritten as:
myVar -= 5;

1.20 Compound Assignment With Augmented Multiplication

The *= operator multiplies a variable by a number.
myVar = myVar * 5;
will multiply myVar by 5. This can be rewritten as:
myVar *= 5;

1.21 Compound Assignment With Augmented Division

The /= operator divides a variable by another number.
myVar = myVar / 5;
Will divide myVar by 5. This can be rewritten as:
myVar /= 5;

1.22 Declare String Variables

Previously we have used the code
var myName = "your name";
"your name" is called a string literal. It is a string because it is a series of zero or more characters enclosed in single or double quotes.

1.23 Escaping Literal Quotes in Strings

When you are defining a string you must start and end with a single or double quote. What happens when you need a literal quote: " or ' inside of your string?
In JavaScript, you can escape a quote from considering it as an end of string quote by placing a backslash (\) in front of the quote.
var sampleStr = "Alan said, \"Peter is learning JavaScript\".";
This signals to JavaScript that the following quote is not the end of the string, but should instead appear inside the string. So if you were to print this to the console, you would get:
Alan said, "Peter is learning JavaScript".

1.24 Quoting Strings with Single Quotes

String values in JavaScript may be written with single or double quotes, as long as you start and end with the same type of quote. Unlike some other programming languages, single and double quotes work the same in JavaScript.
“‘js
doubleQuoteStr = "This is a string";
singleQuoteStr = 'This is also a string';
“‘
The reason why you might want to use one type of quote over the other is if you want to use both in a string. This might happen if you want to save a conversation in a string and have the conversation in quotes. Another use for it would be saving an <a> tag with various attributes in quotes, all within a string.
“‘js
conversation = 'Finn exclaims to Jake, "Algebraic!"';
“‘
However, this becomes a problem if you need to use the outermost quotes within it. Remember, a string has the same kind of quote at the beginning and end. But if you have that same quote somewhere in the middle, the string will stop early and throw an error.
“‘js

goodStr = 'Jake asks Finn, "Hey, let\'s go on an adventure?"';
badStr = 'Finn responds, "Let's go!"'; // Throws an error
‘‘‘
In the goodStr above, you can use both quotes safely by using the backslash \ as an escape character.
NoteThe backslash \ should not be confused with the forward slash /. They do not do the same thing.

1.25 Escape Sequences in Strings

Quotes are not the only characters that can be escaped inside a string. There are two reasons to use escaping characters:To allow you to use characters you may not otherwise be able to type out, such as a carriage return.To allow you to represent multiple quotes in a string without JavaScript misinterpreting what you mean.We learned this in the previous challenge.
CodeOutput\'single quote\"double quote\\backslash\nnewline\rcarriage return\ttab\bword boundary\fform feed
Note that the backslash itself must be escaped in order to display as a backslash.

1.26 Concatenating Strings with Plus Operator

In JavaScript, when the + operator is used with a String value, it is called the concatenation operator. You can build a new string out of other strings by concatenating them together.
Example
‘‘‘js
'My name is Alan,' + ' I concatenate.'
‘‘‘
NoteWatch out for spaces. Concatenation does not add spaces between concatenated strings, so you'll need to add them yourself.
Example:
‘‘‘js
var ourStr = "I come first. " + "I come second.";
// ourStr is "I come first. I come second."
‘‘‘

1.27 Concatenating Strings with the Plus Equals Operator

We can also use the += operator to concatenate a string onto the end of an existing string variable. This can be very helpful to break a long string over several lines.
NoteWatch out for spaces. Concatenation does not add spaces between concatenated strings, so you'll need to add them yourself.
Example:
‘‘‘js
var ourStr = "I come first. ";
ourStr += "I come second.";
// ourStr is now "I come first. I come second."
‘‘‘

1.28 Constructing Strings with Variables

Sometimes you will need to build a string, Mad Libs style. By using the concatenation operator (+), you can insert one or more variables into a string you're building.
Example:
‘‘‘js

var ourName = "freeCodeCamp";
var ourStr = "Hello, our name is " + ourName + ", how are you?";
// ourStr is now "Hello, our name is freeCodeCamp, how are you?"
‘‘‘

1.29 Appending Variables to Strings

Just as we can build a string over multiple lines out of string literals, we can also append variables to a string using the plus equals (+=) operator.
Example:
‘‘‘js
var anAdjective = "awesome!";
var ourStr = "freeCodeCamp is ";
ourStr += anAdjective;
// ourStr is now "freeCodeCamp is awesome!"
‘‘‘

1.30 Find the Length of a String

You can find the length of a String value by writing .length after the string variable or string literal.
"Alan Peter".length; // 10
For example, if we created a variable var firstName = "Charles", we could find out how long the string "Charles" is by using the firstName.length property.

1.31 Use Bracket Notation to Find the First Character in a String

Bracket notation is a way to get a character at a specific index within a string.
Most modern programming languages, like JavaScript, don't start counting at 1 like humans do. They start at 0. This is referred to as Zero-based indexing.
For example, the character at index 0 in the word "Charles" is "C". So if var firstName = "Charles", you can get the value of the first letter of the string by using firstName[0].
Example:
‘‘‘js
var firstName = "Charles";
var firstLetter = firstName[0]; // firstLetter is "C"
‘‘‘

1.32 Understand String Immutability

In JavaScript, String values are immutable, which means that they cannot be altered once created.
For example, the following code:
‘‘‘js
var myStr = "Bob";
myStr[0] = "J";
‘‘‘
cannot change the value of myStr to "Job", because the contents of myStr cannot be altered. Note that this does not mean that myStr cannot be changed, just that the individual characters of a string literal cannot be changed. The only way to change myStr would be to assign it with a new string, like this:
‘‘‘js
var myStr = "Bob";
myStr = "Job";

```

## 1.33 Use Bracket Notation to Find the Nth Character in a String

You can also use bracket notation to get the character at other positions within a string.
Remember that computers start counting at 0, so the first character is actually the zeroth character.
Example:
```js
var firstName = "Ada";
var secondLetterOfFirstName = firstName[1]; // secondLetterOfFirstName is "d"
```

1.34 Use Bracket Notation to Find the Last Character in a String

In order to get the last letter of a string, you can subtract one from the string's length.
For example, if var firstName = "Charles", you can get the value of the last letter of the string by using firstName[firstName.length - 1].
Example:
```js
var firstName = "Charles";
var lastLetter = firstName[firstName.length - 1]; // lastLetter is "s"
```

1.35 Use Bracket Notation to Find the Nth-to-Last Character in a String

You can use the same principle we just used to retrieve the last character in a string to retrieve the Nth-to-last character.
For example, you can get the value of the third-to-last letter of the var firstName = "Charles" string by using firstName[firstName.length - 3]
Example:
```js
var firstName = "Charles";
var thirdToLastLetter = firstName[firstName.length - 3]; // thirdToLastLetter is "l"
```

1.36 Word Blanks

We will now use our knowledge of strings to build a "Mad Libs" style word game we're calling "Word Blanks". You will create an (optionally humorous) "Fill in the Blanks" style sentence.
In a "Mad Libs" game, you are provided sentences with some missing words, like nouns, verbs, adjectives and adverbs. You then fill in the missing pieces with words of your choice in a way that the completed sentence makes sense.
Consider this sentence - "It was really _____, and we _____ ourselves _____". This sentence has three missing pieces- an adjective, a verb and an adverb, and we can add words of our choice to complete it. We can then assign the completed sentence to a variable as follows:
```js
var sentence = "It was really " + "hot" + ", and we " + "laughed" + " ourselves " + "silly" + ".";
```
```

1.37 Store Multiple Values in one Variable using JavaScript Arrays

With JavaScript array variables, we can store several pieces of data in one place.
You start an array declaration with an opening square bracket, end it with a closing square bracket, and put a comma between each entry, like this:
var sandwich = ["peanut butter", "jelly", "bread"].

1.38 Nest one Array within Another Array

You can also nest arrays within other arrays, like below:
“‘js
[["Bulls", 23], ["White Sox", 45]]
“‘
This is also called a multi-dimensional array.

1.39 Access Array Data with Indexes

We can access the data inside arrays using indexes.
Array indexes are written in the same bracket notation that strings use, except that instead of specifying a character, they are specifying an entry in the array. Like strings, arrays use zero-based indexing, so the first element in an array has an index of 0.
Example
“‘js
var array = [50,60,70];
array[0]; // equals 50
var data = array[1]; // equals 60
“‘
NoteThere shouldn’t be any spaces between the array name and the square brackets, like array [0]. Although JavaScript is able to process this correctly, this may confuse other programmers reading your code.

1.40 Modify Array Data With Indexes

Unlike strings, the entries of arrays are mutable and can be changed freely.
Example
“‘js
var ourArray = [50,40,30];
ourArray[0] = 15; // equals [15,40,30]
“‘
NoteThere shouldn’t be any spaces between the array name and the square brackets, like array [0]. Although JavaScript is able to process this correctly, this may confuse other programmers reading your code.

1.41 Access Multi-Dimensional Arrays With Indexes

One way to think of a multi-dimensional array, is as an array of arrays. When you use brackets to access your array, the first set of brackets refers to the entries in the outer-most (the first level) array, and each additional pair of brackets refers to the next level of entries inside.
Example
“‘js
var arr = [
[1,2,3],
[4,5,6],

[7,8,9],
[[10,11,12], 13, 14]
];
arr[3]; // equals [[10,11,12], 13, 14]
arr[3][0]; // equals [10,11,12]
arr[3][0][1]; // equals 11
‘‘‘

NoteThere shouldn’t be any spaces between the array name and the square brackets, like ‘array [0][0]‘ and even this ‘array [0] [0]‘ is not allowed. Although JavaScript is able to process this correctly, this may confuse other programmers reading your code.

1.42 Manipulate Arrays With push()

An easy way to append data to the end of an array is via the push() function.
.push() takes one or more parameters and "pushes" them onto the end of the array.
Examples:
“‘js
var arr1 = [1,2,3];
arr1.push(4);
// arr1 is now [1,2,3,4]
var arr2 = ["Stimpson", "J", "cat"];
arr2.push(["happy", "joy"]);
// arr2 now equals ["Stimpson", "J", "cat", ["happy", "joy"]]
‘‘‘

1.43 Manipulate Arrays With pop()

Another way to change the data in an array is with the .pop() function.
.pop() is used to "pop" a value off of the end of an array. We can store this "popped off" value by assigning it to a variable. In other words, .pop() removes the last element from an array and returns that element.
Any type of entry can be "popped" off of an array - numbers, strings, even nested arrays.
“‘js
var threeArr = [1, 4, 6];
var oneDown = threeArr.pop();
console.log(oneDown); // Returns 6
console.log(threeArr); // Returns [1, 4]
‘‘‘

1.44 Manipulate Arrays With shift()

pop() always removes the last element of an array. What if you want to remove the first?
That’s where .shift() comes in. It works just like .pop(), except it removes the first element instead of the last.
Example:
“‘js
var ourArray = ["Stimpson", "J", ["cat"]];
var removedFromOurArray = ourArray.shift();
// removedFromOurArray now equals "Stimpson" and ourArray now equals ["J", ["cat"]].
‘‘‘

1.45 Manipulate Arrays With unshift()

Not only can you shift elements off of the beginning of an array, you can also unshift elements to the beginning of an array i.e. add elements in front of the array.
.unshift() works exactly like .push(), but instead of adding the element at the end of the array, unshift() adds the element at the beginning of the array.
Example:
“‘js
var ourArray = ["Stimpson", "J", "cat"];
ourArray.shift(); // ourArray now equals ["J", "cat"]
ourArray.unshift("Happy");
// ourArray now equals ["Happy", "J", "cat"]
“‘

1.46 Shopping List

Create a shopping list in the variable myList. The list should be a multi-dimensional array containing several sub-arrays.
The first element in each sub-array should contain a string with the name of the item. The second element should be a number representing the quantity i.e.
["Chocolate Bar", 15]
There should be at least 5 sub-arrays in the list.

1.47 Write Reusable JavaScript with Functions

In JavaScript, we can divide up our code into reusable parts called functions.
Here’s an example of a function:
“‘js
function functionName() {
console.log("Hello World");
}
“‘
You can call or invoke this function by using its name followed by parentheses, like this:
functionName();
Each time the function is called it will print out the message "Hello World" on the dev console. All of the code between the curly braces will be executed every time the function is called.

1.48 Passing Values to Functions with Arguments

Parameters are variables that act as placeholders for the values that are to be input to a function when it is called. When a function is defined, it is typically defined along with one or more parameters. The actual values that are input (or "passed") into a function when it is called are known as arguments.
Here is a function with two parameters, param1 and param2:
“‘js
function testFun(param1, param2) {
console.log(param1, param2);
}
“‘
Then we can call testFun:
testFun("Hello", "World");
We have passed two arguments, "Hello" and "World". Inside the function, param1 will equal "Hello" and

param2 will equal "World". Note that you could call testFun again with different arguments and the parameters would take on the value of the new arguments.

1.49 Global Scope and Functions

In JavaScript, scope refers to the visibility of variables. Variables which are defined outside of a function block have Global scope. This means, they can be seen everywhere in your JavaScript code.
Variables which are used without the var keyword are automatically created in the global scope. This can create unintended consequences elsewhere in your code or when running a function again. You should always declare your variables with var.

1.50 Local Scope and Functions

Variables which are declared within a function, as well as the function parameters have local scope. That means, they are only visible within that function.
Here is a function myTest with a local variable called loc.
“‘js
function myTest() {
var loc = "foo";
console.log(loc);
}
myTest(); // logs "foo"
console.log(loc); // loc is not defined
“‘
loc is not defined outside of the function.

1.51 Global vs. Local Scope in Functions

It is possible to have both local and global variables with the same name. When you do this, the local variable takes precedence over the global variable.
In this example:
“‘js
var someVar = "Hat";
function myFun() {
var someVar = "Head";
return someVar;
}
“‘
The function myFun will return "Head" because the local version of the variable is present.

1.52 Return a Value from a Function with Return

We can pass values into a function with arguments. You can use a return statement to send a value back out of a function.
Example
“‘js
function plusThree(num) {
return num + 3;
}
var answer = plusThree(5); // 8
“‘

plusThree takes an argument for num and returns a value equal to num + 3.

1.53 Understanding Undefined Value returned from a Function

A function can include the return statement but it does not have to. In the case that the function doesn't have a return statement, when you call it, the function processes the inner code but the returned value is undefined.
Example
“‘js
var sum = 0;
function addSum(num) {
sum = sum + num;
}
addSum(3); // sum will be modified but returned value is undefined
“‘
addSum is a function without a return statement. The function will change the global sum variable but the returned value of the function is undefined.

1.54 Assignment with a Returned Value

If you'll recall from our discussion of Storing Values with the Assignment Operator, everything to the right of the equal sign is resolved before the value is assigned. This means we can take the return value of a function and assign it to a variable.
Assume we have pre-defined a function sum which adds two numbers together, then:
ourSum = sum(5, 12);
will call sum function, which returns a value of 17 and assigns it to ourSum variable.

1.55 Stand in Line

In Computer Science a queue is an abstract Data Structure where items are kept in order. New items can be added at the back of the queue and old items are taken off from the front of the queue.
Write a function nextInLine which takes an array (arr) and a number (item) as arguments.
Add the number to the end of the array, then remove the first element of the array.
The nextInLine function should then return the element that was removed.

1.56 Understanding Boolean Values

Another data type is the Boolean. Booleans may only be one of two values: true or false. They are basically little on-off switches, where true is "on" and false is "off." These two states are mutually exclusive.
NoteBoolean values are never written with quotes. The strings "true" and "false" are not Boolean and have no special meaning in JavaScript.

1.57 Use Conditional Logic with If Statements

If statements are used to make decisions in code. The keyword if tells JavaScript to execute the code in the curly braces under certain conditions, defined in the parentheses. These conditions are known as Boolean conditions and they may only be true or false.
When the condition evaluates to true, the program executes the statement inside the curly braces. When the Boolean condition evaluates to false, the statement inside the curly braces will not execute.
Pseudocode

if (condition is true) { statement is executed}
Example
“‘js
function test (myCondition) {
if (myCondition) {
return "It was true";
}
return "It was false";
}
test(true); // returns "It was true"
test(false); // returns "It was false"
“‘
When test is called with a value of true, the if statement evaluates myCondition to see if it is true or not. Since it is true, the function returns "It was true". When we call test with a value of false, myCondition is not true and the statement in the curly braces is not executed and the function returns "It was false".

1.58 Comparison with the Equality Operator

There are many comparison operators in JavaScript. All of these operators return a boolean true or false value.
The most basic operator is the equality operator ==. The equality operator compares two values and returns true if they’re equivalent or false if they are not. Note that equality is different from assignment (=), which assigns the value on the right of the operator to a variable on the left.
“‘js
function equalityTest(myVal) {
if (myVal == 10) {
return "Equal";
}
return "Not Equal";
}
“‘
If myVal is equal to 10, the equality operator returns true, so the code in the curly braces will execute, and the function will return "Equal". Otherwise, the function will return "Not Equal".
In order for JavaScript to compare two different data types (for example, numbers and strings), it must convert one type to another. This is known as "Type Coercion". Once it does, however, it can compare terms as follows:
“‘js
1 == 1 // true
1 == 2 // false
1 == ’1’ // true
"3" == 3 // true
“‘

1.59 Comparison with the Strict Equality Operator

Strict equality (===) is the counterpart to the equality operator (==). However, unlike the equality operator, which attempts to convert both values being compared to a common type, the strict equality operator does not perform a type conversion.
If the values being compared have different types, they are considered unequal, and the strict equality operator will return false.
Examples
“‘js

3 === 3 // true
3 === '3' // false
‘‘‘
In the second example, 3 is a Number type and '3' is a String type.

1.60 Practice comparing different values

In the last two challenges, we learned about the equality operator (==) and the strict equality operator (===). Let's do a quick review and practice using these operators some more.
If the values being compared are not of the same type, the equality operator will perform a type conversion, and then evaluate the values. However, the strict equality operator will compare both the data type and value as-is, without converting one type to the other.
Examples
“‘js
3 == '3' // returns true because JavaScript performs type conversion from string to number
3 === '3' // returns false because the types are different and type conversion is not performed
‘‘‘
NoteIn JavaScript, you can determine the type of a variable or a value with the typeof operator, as follows:
“‘js
typeof 3 // returns 'number'
typeof '3' // returns 'string'
‘‘‘

1.61 Comparison with the Inequality Operator

The inequality operator (!=) is the opposite of the equality operator. It means "Not Equal" and returns false where equality would return true and vice versa. Like the equality operator, the inequality operator will convert data types of values while comparing.
Examples
“‘js
1 != 2 // true
1 != "1" // false
1 != '1' // false
1 != true // false
0 != false // false
‘‘‘

1.62 Comparison with the Strict Inequality Operator

The strict inequality operator (!==) is the logical opposite of the strict equality operator. It means "Strictly Not Equal" and returns false where strict equality would return true and vice versa. Strict inequality will not convert data types.
Examples
“‘js
3 !== 3 // false
3 !== '3' // true
4 !== 3 // true
‘‘‘

1.63 Comparison with the Greater Than Operator

The greater than operator (>) compares the values of two numbers. If the number to the left is greater than the number to the right, it returns true. Otherwise, it returns false.
Like the equality operator, greater than operator will convert data types of values while comparing.
Examples
“‘js
5 > 3 // true
7 > ’3’ // true
2 > 3 // false
’1’ > 9 // false
“‘

1.64 Comparison with the Greater Than Or Equal To Operator

The greater than or equal to operator (>=) compares the values of two numbers. If the number to the left is greater than or equal to the number to the right, it returns true. Otherwise, it returns false.
Like the equality operator, greater than or equal to operator will convert data types while comparing.
Examples
“‘js
6 >= 6 // true
7 >= ’3’ // true
2 >= 3 // false
’7’ >= 9 // false
“‘

1.65 Comparison with the Less Than Operator

The less than operator (<) compares the values of two numbers. If the number to the left is less than the number to the right, it returns true. Otherwise, it returns false. Like the equality operator, less than operator converts data types while comparing.
Examples
“‘js
2 < 5 // true
’3’ < 7 // true
5 < 5 // false
3 < 2 // false
’8’ < 4 // false
“‘

1.66 Comparison with the Less Than Or Equal To Operator

The less than or equal to operator (<=) compares the values of two numbers. If the number to the left is less than or equal to the number to the right, it returns true. If the number on the left is greater than the number on the right, it returns false. Like the equality operator, less than or equal to converts data types.
Examples
“‘js
4 <= 5 // true
’7’ <= 7 // true
5 <= 5 // true
3 <= 2 // false
’8’ <= 4 // false

```

## 1.67 Comparisons with the Logical And Operator

Sometimes you will need to test more than one thing at a time. The logical and operator (&&) returns true if and only if the operands to the left and right of it are true.
The same effect could be achieved by nesting an if statement inside another if:
```js
if (num > 5) {
if (num < 10) {
return "Yes";
}
}
return "No";
```
will only return "Yes" if num is greater than 5 and less than 10. The same logic can be written as:
```js
if (num > 5 && num < 10) {
return "Yes";
}
return "No";
```

1.68 Comparisons with the Logical Or Operator

The logical or operator (||) returns true if either of the operands is true. Otherwise, it returns false.
The logical or operator is composed of two pipe symbols: (||). This can typically be found between your Backspace and Enter keys.
The pattern below should look familiar from prior waypoints:
```js
if (num > 10) {
return "No";
}
if (num < 5) {
return "No";
}
return "Yes";
```
will return "Yes" only if num is between 5 and 10 (5 and 10 included). The same logic can be written as:
```js
if (num > 10 || num < 5) {
return "No";
}
return "Yes";
```

1.69 Introducing Else Statements

When a condition for an if statement is true, the block of code following it is executed. What about when that condition is false? Normally nothing would happen. With an else statement, an alternate block of code can be executed.
```js

```
if (num > 10) {
return "Bigger than 10";
} else {
return "10 or Less";
}
```

1.70 Introducing Else If Statements

If you have multiple conditions that need to be addressed, you can chain if statements together with else if statements.

```js
if (num > 15) {
return "Bigger than 15";
} else if (num < 5) {
return "Smaller than 5";
} else {
return "Between 5 and 15";
}
```

1.71 Logical Order in If Else Statements

Order is important in if, else if statements.
The function is executed from top to bottom so you will want to be careful of what statement comes first.
Take these two functions as an example.
Here's the first:

```js
function foo(x) {
if (x < 1) {
return "Less than one";
} else if (x < 2) {
return "Less than two";
} else {
return "Greater than or equal to two";
}
}
```

And the second just switches the order of the statements:

```js
function bar(x) {
if (x < 2) {
return "Less than two";
} else if (x < 1) {
return "Less than one";
} else {
return "Greater than or equal to two";
}
}
```

While these two functions look nearly identical if we pass a number to both we get different outputs.

```js
```

foo(0) // "Less than one"
bar(0) // "Less than two"
‘‘‘

1.72 Chaining If Else Statements

if/else statements can be chained together for complex logic. Here is pseudocode of multiple chained if / else if statements:
“‘js
if (condition1) {
statement1
} else if (condition2) {
statement2
} else if (condition3) {
statement3
. . .
} else {
statementN
}
‘‘‘

1.73 Golf Code

In the game of golf each hole has a par meaning the average number of strokes a golfer is expected to make in order to sink the ball in a hole to complete the play. Depending on how far above or below par your strokes are, there is a different nickname.
Your function will be passed par and strokes arguments. Return the correct string according to this table which lists the strokes in order of priority; top (highest) to bottom (lowest):
StrokesReturn1"Hole-in-one!"<= par - 2"Eagle"par - 1"Birdie"par"Par"par + 1"Bogey"par + 2"Double Bogey">= par + 3"Go Home!"
par and strokes will always be numeric and positive. We have added an array of all the names for your convenience.

1.74 Selecting from Many Options with Switch Statements

If you have many options to choose from, use a switch statement. A switch statement tests a value and can have many case statements which define various possible values. Statements are executed from the first matched case value until a break is encountered.
Here is an example of a switch statement:
“‘js
switch(lowercaseLetter) {
case "a":
console.log("A");
break;
case "b":
console.log("B");
break;
}
‘‘‘
case values are tested with strict equality (===). The break tells JavaScript to stop executing statements. If the break is omitted, the next statement will be executed.

1.75 Adding a Default Option in Switch Statements

In a switch statement you may not be able to specify all possible values as case statements. Instead, you can add the default statement which will be executed if no matching case statements are found. Think of it like the final else statement in an if/else chain.
A default statement should be the last case.
“‘js
switch (num) {
case value1:
statement1;
break;
case value2:
statement2;
break;
...
default:
defaultStatement;
break;
}
“‘

1.76 Multiple Identical Options in Switch Statements

If the break statement is omitted from a switch statement’s case, the following case statement(s) are executed until a break is encountered. If you have multiple inputs with the same output, you can represent them in a switch statement like this:
“‘js
switch(val) {
case 1:
case 2:
case 3:
result = "1, 2, or 3";
break;
case 4:
result = "4 alone";
}
“‘
Cases for 1, 2, and 3 will all produce the same result.

1.77 Replacing If Else Chains with Switch

If you have many options to choose from, a switch statement can be easier to write than many chained if/else if statements. The following:
“‘js
if (val === 1) {
answer = "a";
} else if (val === 2) {
answer = "b";
} else {
answer = "c";
}
“‘
can be replaced with:

```js
switch(val) {
case 1:
answer = "a";
break;
case 2:
answer = "b";
break;
default:
answer = "c";
}
```

1.78 Returning Boolean Values from Functions

You may recall from Comparison with the Equality Operator that all comparison operators return a boolean true or false value.
Sometimes people use an if/else statement to do a comparison, like this:
```js
function isEqual(a,b) {
if (a === b) {
return true;
} else {
return false;
}
}
```
But there's a better way to do this. Since === returns true or false, we can return the result of the comparison:
```js
function isEqual(a,b) {
return a === b;
}
```

1.79 Return Early Pattern for Functions

When a return statement is reached, the execution of the current function stops and control returns to the calling location.
Example
```js
function myFun() {
console.log("Hello");
return "World";
console.log("byebye")
}
myFun();
```
The above outputs "Hello" to the console, returns "World", but "byebye" is never output, because the function exits at the return statement.

1.80 Counting Cards

In the casino game Blackjack, a player can gain an advantage over the house by keeping track of the relative number of high and low cards remaining in the deck. This is called Card Counting.
Having more high cards remaining in the deck favors the player. Each card is assigned a value according to the table below. When the count is positive, the player should bet high. When the count is zero or negative, the player should bet low.
Count ChangeCards+12, 3, 4, 5, 607, 8, 9-110, 'J', 'Q', 'K', 'A'
You will write a card counting function. It will receive a card parameter, which can be a number or a string, and increment or decrement the global count variable according to the card's value (see table). The function will then return a string with the current count and the string Bet if the count is positive, or Hold if the count is zero or negative. The current count and the player's decision (Bet or Hold) should be separated by a single space.
Example Output-3 Hold5 Bet
HintDo NOT reset count to 0 when value is 7, 8, or 9.Do NOT return an array.Do NOT include quotes (single or double) in the output.

1.81 Build JavaScript Objects

You may have heard the term object before.
Objects are similar to arrays, except that instead of using indexes to access and modify their data, you access the data in objects through what are called properties.
Objects are useful for storing data in a structured way, and can represent real world objects, like a cat.
Here's a sample cat object:
“‘js
var cat = {
"name": "Whiskers",
"legs": 4,
"tails": 1,
"enemies": ["Water", "Dogs"]
};
“‘
In this example, all the properties are stored as strings, such as - "name", "legs", and "tails". However, you can also use numbers as properties. You can even omit the quotes for single-word string properties, as follows:
“‘js
var anotherObject = {
make: "Ford",
5: "five",
"model": "focus"
};
“‘
However, if your object has any non-string properties, JavaScript will automatically typecast them as strings.

1.82 Accessing Object Properties with Dot Notation

There are two ways to access the properties of an object: dot notation (.) and bracket notation ([]), similar to an array.
Dot notation is what you use when you know the name of the property you're trying to access ahead of time.
Here is a sample of using dot notation (.) to read an object's property:
“‘js
var myObj = {
prop1: "val1",
prop2: "val2"

};
var prop1val = myObj.prop1; // val1
var prop2val = myObj.prop2; // val2
‘‘‘

1.83 Accessing Object Properties with Bracket Notation

The second way to access the properties of an object is bracket notation ([]). If the property of the object you are trying to access has a space in its name, you will need to use bracket notation.
However, you can still use bracket notation on object properties without spaces.
Here is a sample of using bracket notation to read an object's property:
“‘js
var myObj = {
"Space Name": "Kirk",
"More Space": "Spock",
"NoSpace": "USS Enterprise"
};
myObj["Space Name"]; // Kirk
myObj['More Space']; // Spock
myObj["NoSpace"]; // USS Enterprise
‘‘‘
Note that property names with spaces in them must be in quotes (single or double).

1.84 Accessing Object Properties with Variables

Another use of bracket notation on objects is to access a property which is stored as the value of a variable. This can be very useful for iterating through an object's properties or when accessing a lookup table.
Here is an example of using a variable to access a property:
“‘js
var dogs = {
Fido: "Mutt", Hunter: "Doberman", Snoopie: "Beagle"
};
var myDog = "Hunter";
var myBreed = dogs[myDog];
console.log(myBreed); // "Doberman"
‘‘‘
Another way you can use this concept is when the property's name is collected dynamically during the program execution, as follows:
“‘js
var someObj = {
propName: "John"
};
function propPrefix(str) {
var s = "prop";
return s + str;
}
var someProp = propPrefix("Name"); // someProp now holds the value 'propName'
console.log(someObj[someProp]); // "John"
‘‘‘
Note that we do not use quotes around the variable name when using it to access the property because we are using the value of the variable, not the name.

1.85 Updating Object Properties

After you've created a JavaScript object, you can update its properties at any time just like you would update any other variable. You can use either dot or bracket notation to update.
For example, let's look at ourDog:
“‘js
var ourDog = {
"name": "Camper",
"legs": 4,
"tails": 1,
"friends": ["everything!"]
};
“‘
Since he's a particularly happy dog, let's change his name to "Happy Camper". Here's how we update his object's name property:
ourDog.name = "Happy Camper"; or
ourDog["name"] = "Happy Camper";
Now when we evaluate ourDog.name, instead of getting "Camper", we'll get his new name, "Happy Camper".

1.86 Add New Properties to a JavaScript Object

You can add new properties to existing JavaScript objects the same way you would modify them.
Here's how we would add a "bark" property to ourDog:
ourDog.bark = "bow-wow";
or
ourDog["bark"] = "bow-wow";
Now when we evaluate ourDog.bark, we'll get his bark, "bow-wow".
Example:
“‘js
var ourDog = {
"name": "Camper",
"legs": 4,
"tails": 1,
"friends": ["everything!"]
};
ourDog.bark = "bow-wow";
“‘

1.87 Delete Properties from a JavaScript Object

We can also delete properties from objects like this:
delete ourDog.bark;
Example:
“‘js
var ourDog = {
"name": "Camper",
"legs": 4,
"tails": 1,
"friends": ["everything!"],
"bark": "bow-wow"
};
delete ourDog.bark;
“‘

After the last line shown above, ourDog looks like:
“‘js
{
"name": "Camper",
"legs": 4,
"tails": 1,
"friends": ["everything!"]
}
“‘

1.88 Using Objects for Lookups

Objects can be thought of as a key/value storage, like a dictionary. If you have tabular data, you can use an object to "lookup" values rather than a switch statement or an if/else chain. This is most useful when you know that your input data is limited to a certain range.
Here is an example of a simple reverse alphabet lookup:
“‘js
var alpha = {
1:"Z",
2:"Y",
3:"X",
4:"W",
...
24:"C",
25:"B",
26:"A"
};
alpha[2]; // "Y"
alpha[24]; // "C"
var value = 2;
alpha[value]; // "Y"
“‘

1.89 Testing Objects for Properties

Sometimes it is useful to check if the property of a given object exists or not. We can use the .hasOwnProperty(propname) method of objects to determine if that object has the given property name. .hasOwnProperty() returns true or false if the property is found or not.
Example
“‘js
var myObj = {
top: "hat",
bottom: "pants"
};
myObj.hasOwnProperty("top"); // true
myObj.hasOwnProperty("middle"); // false
“‘

1.90 Manipulating Complex Objects

Sometimes you may want to store data in a flexible Data Structure. A JavaScript object is one way to handle flexible data. They allow for arbitrary combinations of strings, numbers, booleans, arrays, functions, and

objects.
Here's an example of a complex data structure:
“‘js
var ourMusic = [
{
"artist": "Daft Punk",
"title": "Homework",
"release_year": 1997,
"formats": [
"CD",
"Cassette",
"LP"
],
"gold": true
}
];
“‘
This is an array which contains one object inside. The object has various pieces of metadata about an album. It also has a nested "formats" array. If you want to add more album records, you can do this by adding records to the top level array.
Objects hold data in a property, which has a key-value format. In the example above, "artist": "Daft Punk" is a property that has a key of "artist" and a value of "Daft Punk".
JavaScript Object Notation or JSON is a related data interchange format used to store data.
“‘json
{
"artist": "Daft Punk",
"title": "Homework",
"release_year": 1997,
"formats": [
"CD",
"Cassette",
"LP"
],
"gold": true
}
“‘
NoteYou will need to place a comma after every object in the array, unless it is the last object in the array.

1.91 Accessing Nested Objects

The sub-properties of objects can be accessed by chaining together the dot or bracket notation.
Here is a nested object:
“‘js
var ourStorage = {
"desk": {
"drawer": "stapler"
},
"cabinet": {
"top drawer": {
"folder1": "a file",
"folder2": "secrets"
},
"bottom drawer": "soda"

}
};
ourStorage.cabinet["top drawer"].folder2; // "secrets"
ourStorage.desk.drawer; // "stapler"
‘‘‘

1.92 Accessing Nested Arrays

As we have seen in earlier examples, objects can contain both nested objects and nested arrays. Similar to accessing nested objects, Array bracket notation can be chained to access nested arrays.
Here is an example of how to access a nested array:
‘‘‘js
var ourPets = [
{
animalType: "cat",
names: [
"Meowzer",
"Fluffy",
"Kit-Cat"
]
},
{
animalType: "dog",
names: [
"Spot",
"Bowser",
"Frankie"
]
}
];
ourPets[0].names[1]; // "Fluffy"
ourPets[1].names[0]; // "Spot"
‘‘‘

1.93 Record Collection

You are given a JSON object representing a part of your musical album collection. Each album has a unique id number as its key and several other properties. Not all albums have complete information.
You start with an ‘updateRecords‘ function that takes an object like ‘collection‘, an ‘id‘, a ‘prop‘ (like ‘artist‘ or ‘tracks‘), and a ‘value‘. Complete the function using the rules below to modify the object passed to the function.
- Your function must always return the entire object.
- If ‘prop‘ isn’t ‘tracks‘ and ‘value‘ isn’t an empty string, update or set that album’s ‘prop‘ to ‘value‘.
- If ‘prop‘ is ‘tracks‘ but the album doesn’t have a ‘tracks‘ property, create an empty array and add ‘value‘ to it.
- If ‘prop‘ is ‘tracks‘ and ‘value‘ isn’t an empty string, add ‘value‘ to the end of the album’s existing ‘tracks‘ array.
- If ‘value‘ is an empty string, delete the given ‘prop‘ property from the album.
Note: A copy of the ‘collection‘ object is used for the tests.

1.94 Iterate with JavaScript While Loops

You can run the same code multiple times by using a loop.
The first type of loop we will learn is called a while loop because it runs "while" a specified condition is true and stops once that condition is no longer true.

```js
var ourArray = [];
var i = 0;
while(i < 5) {
ourArray.push(i);
i++;
}
```

In the code example above, the while loop will execute 5 times and append the numbers 0 through 4 to ourArray.
Let's try getting a while loop to work by pushing values to an array.

1.95 Iterate with JavaScript For Loops

You can run the same code multiple times by using a loop.
The most common type of JavaScript loop is called a for loop because it runs "for" a specific number of times.
For loops are declared with three optional expressions separated by semicolons:
for ([initialization]; [condition]; [final-expression])
The initialization statement is executed one time only before the loop starts. It is typically used to define and setup your loop variable.
The condition statement is evaluated at the beginning of every loop iteration and will continue as long as it evaluates to true. When condition is false at the start of the iteration, the loop will stop executing. This means if condition starts as false, your loop will never execute.
The final-expression is executed at the end of each loop iteration, prior to the next condition check and is usually used to increment or decrement your loop counter.
In the following example we initialize with i = 0 and iterate while our condition i < 5 is true. We'll increment i by 1 in each loop iteration with i++ as our final-expression.

```js
var ourArray = [];
for (var i = 0; i < 5; i++) {
ourArray.push(i);
}
```

ourArray will now contain [0,1,2,3,4].

1.96 Iterate Odd Numbers With a For Loop

For loops don't have to iterate one at a time. By changing our final-expression, we can count by even numbers.
We'll start at i = 0 and loop while i < 10. We'll increment i by 2 each loop with i += 2.

```js
var ourArray = [];
for (var i = 0; i < 10; i += 2) {
ourArray.push(i);
}
```

ourArray will now contain [0,2,4,6,8].

Let's change our initialization so we can count by odd numbers.

1.97 Count Backwards With a For Loop

A for loop can also count backwards, so long as we can define the right conditions.
In order to count backwards by twos, we'll need to change our initialization, condition, and final-expression. We'll start at i = 10 and loop while i > 0. We'll decrement i by 2 each loop with i -= 2.
“‘js
var ourArray = [];
for (var i = 10; i > 0; i -= 2) {
ourArray.push(i);
}
“‘
ourArray will now contain [10,8,6,4,2].
Let's change our initialization and final-expression so we can count backward by twos by odd numbers.

1.98 Iterate Through an Array with a For Loop

A common task in JavaScript is to iterate through the contents of an array. One way to do that is with a for loop. This code will output each element of the array arr to the console:
“‘js
var arr = [10, 9, 8, 7, 6];
for (var i = 0; i < arr.length; i++) {
console.log(arr[i]);
}
“‘
Remember that arrays have zero-based indexing, which means the last index of the array is length - 1. Our condition for this loop is i < arr.length, which stops the loop when i is equal to length. In this case the last iteration is i === 4 i.e. when i becomes equal to arr.length and outputs 6 to the console.

1.99 Nesting For Loops

If you have a multi-dimensional array, you can use the same logic as the prior waypoint to loop through both the array and any sub-arrays. Here is an example:
“‘js
var arr = [
[1,2], [3,4], [5,6]
];
for (var i=0; i < arr.length; i++) {
for (var j=0; j < arr[i].length; j++) {
console.log(arr[i][j]);
}
}
“‘
This outputs each sub-element in arr one at a time. Note that for the inner loop, we are checking the .length of arr[i], since arr[i] is itself an array.

1.100 Iterate with JavaScript Do...While Loops

The next type of loop you will learn is called a do...while loop. It is called a do...while loop because it will first do one pass of the code inside the loop no matter what, and then continue to run the loop while the

specified condition evaluates to true.
“‘js
var ourArray = [];
var i = 0;
do {
ourArray.push(i);
i++;
} while (i < 5);
“‘
The example above behaves similar to other types of loops, and the resulting array will look like [0, 1, 2, 3, 4]. However, what makes the do...while different from other loops is how it behaves when the condition fails on the first check. Let's see this in action:
Here is a regular while loop that will run the code in the loop as long as i < 5:
“‘js
var ourArray = [];
var i = 5;
while (i < 5) {
ourArray.push(i);
i++;
}
“‘
In this example, we initialize the value of ourArray to an empty array and the value of i to 5. When we execute the while loop, the condition evaluates to false because i is not less than 5, so we do not execute the code inside the loop. The result is that ourArray will end up with no values added to it, and it will still look like [] when all of the code in the example above has completed running.
Now, take a look at a do...while loop:
“‘js
var ourArray = [];
var i = 5;
do {
ourArray.push(i);
i++;
} while (i < 5);
“‘
In this case, we initialize the value of i to 5, just like we did with the while loop. When we get to the next line, there is no condition to evaluate, so we go to the code inside the curly braces and execute it. We will add a single element to the array and then increment i before we get to the condition check. When we finally evaluate the condition i < 5 on the last line, we see that i is now 6, which fails the conditional check, so we exit the loop and are done. At the end of the above example, the value of ourArray is [5].
Essentially, a do...while loop ensures that the code inside the loop will run at least once.
Let's try getting a do...while loop to work by pushing values to an array.

1.101 Replace Loops using Recursion

Recursion is the concept that a function can be expressed in terms of itself. To help understand this, start by thinking about the following task: multiply the first n elements of an array to create the product of those elements. Using a for loop, you could do this:
“‘js
function multiply(arr, n) {
var product = 1;
for (var i = 0; i < n; i++) {
product *= arr[i];
}

return product;
}
‘‘‘
However, notice that multiply(arr, n) == multiply(arr, n - 1) * arr[n - 1]. That means you can rewrite multiply in terms of itself and never need to use a loop.
‘‘‘js
function multiply(arr, n) {
if (n <= 0) {
return 1;
} else {
return multiply(arr, n - 1) * arr[n - 1];
}
}
‘‘‘
The recursive version of multiply breaks down like this. In the base case, where n <= 0, it returns 1. For larger values of n, it calls itself, but with n - 1. That function call is evaluated in the same way, calling multiply again until n <= 0. At this point, all the functions can return and the original multiply returns the answer.
Note: Recursive functions must have a base case when they return without calling the function again (in this example, when n <= 0), otherwise they can never finish executing.

1.102 Profile Lookup

We have an array of objects representing different people in our contacts lists.
A lookUpProfile function that takes name and a property (prop) as arguments has been pre-written for you.
The function should check if name is an actual contact's firstName and the given property (prop) is a property of that contact.
If both are true, then return the "value" of that property.
If name does not correspond to any contacts then return "No such contact".
If prop does not correspond to any valid properties of a contact found to match name then return "No such property".

1.103 Generate Random Fractions with JavaScript

Random numbers are useful for creating random behavior.
JavaScript has a Math.random() function that generates a random decimal number between 0 (inclusive) and not quite up to 1 (exclusive). Thus Math.random() can return a 0 but never quite return a 1
NoteLike Storing Values with the Equal Operator, all function calls will be resolved before the return executes, so we can return the value of the Math.random() function.

1.104 Generate Random Whole Numbers with JavaScript

It's great that we can generate random decimal numbers, but it's even more useful if we use it to generate random whole numbers.
Use Math.random() to generate a random decimal.Multiply that random decimal by 20.Use another function, Math.floor() to round the number down to its nearest whole number.
Remember that Math.random() can never quite return a 1 and, because we're rounding down, it's impossible to actually get 20. This technique will give us a whole number between 0 and 19.
Putting everything together, this is what our code looks like:
Math.floor(Math.random() * 20);
We are calling Math.random(), multiplying the result by 20, then passing the value to Math.floor() function

to round the value down to the nearest whole number.

1.105 Generate Random Whole Numbers within a Range

Instead of generating a random whole number between zero and a given number like we did before, we can generate a random whole number that falls within a range of two specific numbers.
To do this, we'll define a minimum number min and a maximum number max.
Here's the formula we'll use. Take a moment to read it and try to understand what this code is doing:
Math.floor(Math.random() * (max - min + 1)) + min

1.106 Use the parseInt Function

The parseInt() function parses a string and returns an integer. Here's an example:
var a = parseInt("007");
The above function converts the string "007" to an integer 7. If the first character in the string can't be converted into a number, then it returns NaN.

1.107 Use the parseInt Function with a Radix

The parseInt() function parses a string and returns an integer. It takes a second argument for the radix, which specifies the base of the number in the string. The radix can be an integer between 2 and 36.
The function call looks like:
parseInt(string, radix);
And here's an example:
var a = parseInt("11", 2);
The radix variable says that "11" is in the binary system, or base 2. This example converts the string "11" to an integer 3.

1.108 Use the Conditional (Ternary) Operator

The conditional operator, also called the ternary operator, can be used as a one line if-else expression.
The syntax is:
condition ? expression-if-true : expression-if-false;
The following function uses an if-else statement to check a condition:
“‘js
function findGreater(a, b) {
if(a > b) {
return "a is greater";
}
else {
return "b is greater";
}
}
“‘
This can be re-written using the conditional operator:
“‘js
function findGreater(a, b) {
return a > b ? "a is greater" : "b is greater";
}
“‘

1.109 Use Multiple Conditional (Ternary) Operators

In the previous challenge, you used a single conditional operator. You can also chain them together to check for multiple conditions.
The following function uses if, else if, and else statements to check multiple conditions:
“‘js
function findGreaterOrEqual(a, b) {
if (a === b) {
return "a and b are equal";
}
else if (a > b) {
return "a is greater";
}
else {
return "b is greater";
}
}
“‘
The above function can be re-written using multiple conditional operators:
“‘js
function findGreaterOrEqual(a, b) {
return (a === b) ? "a and b are equal"
: (a > b) ? "a is greater"
: "b is greater";
}
“‘
It is considered best practice to format multiple conditional operators such that each condition is on a separate line, as shown above. Using multiple conditional operators without proper indentation may make your code hard to read. For example:
“‘js
function findGreaterOrEqual(a, b) {
return (a === b) ? "a and b are equal" : (a > b) ? "a is greater" : "b is greater";
}
“‘

1.110 Use Recursion to Create a Countdown

In a [previous challenge](/learn/javascript-algorithms-and-data-structures/basic-javascript/replace-loops-using-recursion), you learned how to use recursion to replace a for loop. Now, let’s look at a more complex function that returns an array of consecutive integers starting with 1 through the number passed to the function.
As mentioned in the previous challenge, there will be a base case. The base case tells the recursive function when it no longer needs to call itself. It is a simple case where the return value is already known. There will also be a recursive call which executes the original function with different arguments. If the function is written correctly, eventually the base case will be reached.
For example, say you want to write a recursive function that returns an array containing the numbers 1 through n. This function will need to accept an argument, n, representing the final number. Then it will need to call itself with progressively smaller values of n until it reaches 1. You could write the function as follows:
“‘javascript
function countup(n) {
if (n < 1) {
return [];
} else {
const countArray = countup(n - 1);

```
countArray.push(n);
return countArray;
}
}
console.log(countup(5)); // [ 1, 2, 3, 4, 5 ]
```

At first, this seems counterintuitive since the value of ‘n‘ decreases, but the values in the final array are increasing. This happens because the push happens last, after the recursive call has returned. At the point where ‘n‘ is pushed into the array, ‘countup(n - 1)‘ has already been evaluated and returned ‘[1, 2, ..., n - 1]‘.

1.111 Use Recursion to Create a Range of Numbers

Continuing from the previous challenge, we provide you another opportunity to create a recursive function to solve a problem.

2 Es6

2.1 Explore Differences Between the var and let Keywords

One of the biggest problems with declaring variables with the var keyword is that you can overwrite variable declarations without an error.
“‘js
var camper = ’James’;
var camper = ’David’;
console.log(camper);
// logs ’David’
“‘
As you can see in the code above, the camper variable is originally declared as James and then overridden to be David.
In a small application, you might not run into this type of problem, but when your code becomes larger, you might accidentally overwrite a variable that you did not intend to overwrite.
Because this behavior does not throw an error, searching and fixing bugs becomes more difficult.
A new keyword called let was introduced in ES6 to solve this potential issue with the var keyword.
If you were to replace var with let in the variable declarations of the code above, the result would be an error.
“‘js
let camper = ’James’;
let camper = ’David’; // throws an error
“‘
This error can be seen in the console of your browser.
So unlike var, when using let, a variable with the same name can only be declared once.
Note the "use strict". This enables Strict Mode, which catches common coding mistakes and "unsafe" actions.
For instance:
“‘js
"use strict";
x = 3.14; // throws an error because x is not declared
“‘

2.2 Compare Scopes of the var and let Keywords

When you declare a variable with the var keyword, it is declared globally, or locally if declared inside a function.
The let keyword behaves similarly, but with some extra features. When you declare a variable with the let keyword inside a block, statement, or expression, its scope is limited to that block, statement, or expression.
For example:
“‘js
var numArray = [];
for (var i = 0; i < 3; i++) {
numArray.push(i);
}
console.log(numArray);
// returns [0, 1, 2]
console.log(i);
// returns 3
“‘
With the var keyword, i is declared globally. So when i++ is executed, it updates the global variable. This code is similar to the following:
“‘js
var numArray = [];
var i;

```
for (i = 0; i < 3; i++) {
numArray.push(i);
}
console.log(numArray);
// returns [0, 1, 2]
console.log(i);
// returns 3
```

This behavior will cause problems if you were to create a function and store it for later use inside a for loop that uses the i variable. This is because the stored function will always refer to the value of the updated global i variable.

```js
var printNumTwo;
for (var i = 0; i < 3; i++) {
if (i === 2) {
printNumTwo = function() {
return i;
};
}
}
console.log(printNumTwo());
// returns 3
```

As you can see, printNumTwo() prints 3 and not 2. This is because the value assigned to i was updated and the printNumTwo() returns the global i and not the value i had when the function was created in the for loop. The let keyword does not follow this behavior:

```js
'use strict';
let printNumTwo;
for (let i = 0; i < 3; i++) {
if (i === 2) {
printNumTwo = function() {
return i;
};
}
}
console.log(printNumTwo());
// returns 2
console.log(i);
// returns "i is not defined"
```

i is not defined because it was not declared in the global scope. It is only declared within the for loop statement. printNumTwo() returned the correct value because three different i variables with unique values (0, 1, and 2) were created by the let keyword within the loop statement.

2.3 Declare a Read-Only Variable with the const Keyword

The keyword let is not the only new way to declare variables. In ES6, you can also declare variables using the const keyword.

const has all the awesome features that let has, with the added bonus that variables declared using const are read-only. They are a constant value, which means that once a variable is assigned with const, it cannot be reassigned.

```js
```

"use strict";
const FAV_PET = "Cats";
FAV_PET = "Dogs"; // returns error
‘‘‘

As you can see, trying to reassign a variable declared with const will throw an error. You should always name variables you don't want to reassign using the const keyword. This helps when you accidentally attempt to reassign a variable that is meant to stay constant. A common practice when naming constants is to use all uppercase letters, with words separated by an underscore.
Note: It is common for developers to use uppercase variable identifiers for immutable values and lowercase or camelCase for mutable values (objects and arrays). In a later challenge you will see an example of a lowercase variable identifier being used for an array.

2.4 Mutate an Array Declared with const

The const declaration has many use cases in modern JavaScript.
Some developers prefer to assign all their variables using const by default, unless they know they will need to reassign the value. Only in that case, they use let.
However, it is important to understand that objects (including arrays and functions) assigned to a variable using const are still mutable. Using the const declaration only prevents reassignment of the variable identifier.
‘‘‘js
"use strict";
const s = [5, 6, 7];
s = [1, 2, 3]; // throws error, trying to assign a const
s[2] = 45; // works just as it would with an array declared with var or let
console.log(s); // returns [5, 6, 45]
‘‘‘

As you can see, you can mutate the object [5, 6, 7] itself and the variable s will still point to the altered array [5, 6, 45]. Like all arrays, the array elements in s are mutable, but because const was used, you cannot use the variable identifier s to point to a different array using the assignment operator.

2.5 Prevent Object Mutation

As seen in the previous challenge, const declaration alone doesn't really protect your data from mutation. To ensure your data doesn't change, JavaScript provides a function Object.freeze to prevent data mutation. Once the object is frozen, you can no longer add, update, or delete properties from it. Any attempt at changing the object will be rejected without an error.
‘‘‘js
let obj = {
name:"FreeCodeCamp",
review:"Awesome"
};
Object.freeze(obj);
obj.review = "bad"; // will be ignored. Mutation not allowed
obj.newProp = "Test"; // will be ignored. Mutation not allowed
console.log(obj);
// { name: "FreeCodeCamp", review:"Awesome"}
‘‘‘

2.6 Use Arrow Functions to Write Concise Anonymous Functions

In JavaScript, we often don't need to name our functions, especially when passing a function as an argument to another function. Instead, we create inline functions. We don't need to name these functions because we

do not reuse them anywhere else.
To achieve this, we often use the following syntax:
“‘js
const myFunc = function() {
const myVar = "value";
return myVar;
}
“‘
ES6 provides us with the syntactic sugar to not have to write anonymous functions this way. Instead, you can use arrow function syntax:
“‘js
const myFunc = () => {
const myVar = "value";
return myVar;
}
“‘
When there is no function body, and only a return value, arrow function syntax allows you to omit the keyword return as well as the brackets surrounding the code. This helps simplify smaller functions into one-line statements:
“‘js
const myFunc = () => "value";
“‘
This code will still return value by default.

2.7 Write Arrow Functions with Parameters

Just like a regular function, you can pass arguments into an arrow function.
“‘js
// doubles input value and returns it
const doubler = (item) => item * 2;
“‘
If an arrow function has a single argument, the parentheses enclosing the argument may be omitted.
“‘js
// the same function, without the argument parentheses
const doubler = item => item * 2;
“‘
It is possible to pass more than one argument into an arrow function.
“‘js
// multiplies the first input value by the second and returns it
const multiplier = (item, multi) => item * multi;
“‘

2.8 Set Default Parameters for Your Functions

In order to help us create more flexible functions, ES6 introduces default parameters for functions.
Check out this code:
“‘js
const greeting = (name = "Anonymous") => "Hello " + name;
console.log(greeting("John")); // Hello John
console.log(greeting()); // Hello Anonymous
“‘
The default parameter kicks in when the argument is not specified (it is undefined). As you can see in the

example above, the parameter name will receive its default value "Anonymous" when you do not provide a value for the parameter. You can add default values for as many parameters as you want.

2.9 Use the Rest Parameter with Function Parameters

In order to help us create more flexible functions, ES6 introduces the rest parameter for function parameters. With the rest parameter, you can create functions that take a variable number of arguments. These arguments are stored in an array that can be accessed later from inside the function.
Check out this code:
“‘js
function howMany(...args) {
return "You have passed " + args.length + " arguments.";
}
console.log(howMany(0, 1, 2)); // You have passed 3 arguments.
console.log(howMany("string", null, [1, 2, 3], { })); // You have passed 4 arguments.
“‘
The rest parameter eliminates the need to check the args array and allows us to apply map(), filter() and reduce() on the parameters array.

2.10 Use the Spread Operator to Evaluate Arrays In-Place

ES6 introduces the spread operator, which allows us to expand arrays and other expressions in places where multiple parameters or elements are expected.
The ES5 code below uses apply() to compute the maximum value in an array:
“‘js
var arr = [6, 89, 3, 45];
var maximus = Math.max.apply(null, arr); // returns 89
“‘
We had to use Math.max.apply(null, arr) because Math.max(arr) returns NaN. Math.max() expects comma-separated arguments, but not an array.
The spread operator makes this syntax much better to read and maintain.
“‘js
const arr = [6, 89, 3, 45];
const maximus = Math.max(...arr); // returns 89
“‘
...arr returns an unpacked array. In other words, it spreads the array.
However, the spread operator only works in-place, like in an argument to a function or in an array literal. The following code will not work:
“‘js
const spreaded = ...arr; // will throw a syntax error
“‘

2.11 Use Destructuring Assignment to Extract Values from Objects

Destructuring assignment is special syntax introduced in ES6, for neatly assigning values taken directly from an object.
Consider the following ES5 code:
“‘js
const user = { name: ’John Doe’, age: 34 };
const name = user.name; // name = ’John Doe’
const age = user.age; // age = 34
“‘

Here's an equivalent assignment statement using the ES6 destructuring syntax:
```js
const { name, age } = user;
// name = 'John Doe', age = 34
```
Here, the name and age variables will be created and assigned the values of their respective values from the user object. You can see how much cleaner this is.
You can extract as many or few values from the object as you want.

2.12 Use Destructuring Assignment to Assign Variables from Objects

Destructuring allows you to assign a new variable name when extracting values. You can do this by putting the new name after a colon when assigning the value.
Using the same object from the last example:
```js
const user = { name: 'John Doe', age: 34 };
```
Here's how you can give new variable names in the assignment:
```js
const { name: userName, age: userAge } = user;
// userName = 'John Doe', userAge = 34
```
You may read it as "get the value of user.name and assign it to a new variable named userName" and so on.

2.13 Use Destructuring Assignment to Assign Variables from Nested Objects

You can use the same principles from the previous two lessons to destructure values from nested objects.
Using an object similar to previous examples:
```js
const user = {
johnDoe: {
age: 34,
email: 'johnDoe@freeCodeCamp.com'
}
};
```
Here's how to extract the values of object properties and assign them to variables with the same name:
```js
const { johnDoe: { age, email }} = user;
```
And here's how you can assign an object properties' values to variables with different names:
```js
const { johnDoe: { age: userAge, email: userEmail }} = user;
```

2.14 Use Destructuring Assignment to Assign Variables from Arrays

ES6 makes destructuring arrays as easy as destructuring objects.
One key difference between the spread operator and array destructuring is that the spread operator unpacks all contents of an array into a comma-separated list. Consequently, you cannot pick or choose which elements you want to assign to variables.
Destructuring an array lets us do exactly that:

“‘js
const [a, b] = [1, 2, 3, 4, 5, 6];
console.log(a, b); // 1, 2
“‘

The variable a is assigned the first value of the array, and b is assigned the second value of the array.
We can also access the value at any index in an array with destructuring by using commas to reach the desired index:
“‘js
const [a, b,,, c] = [1, 2, 3, 4, 5, 6];
console.log(a, b, c); // 1, 2, 5
“‘

2.15 Use Destructuring Assignment with the Rest Parameter to Reassign Array Elements

In some situations involving array destructuring, we might want to collect the rest of the elements into a separate array.
The result is similar to Array.prototype.slice(), as shown below:
“‘js
const [a, b, ...arr] = [1, 2, 3, 4, 5, 7];
console.log(a, b); // 1, 2
console.log(arr); // [3, 4, 5, 7]
“‘

Variables a and b take the first and second values from the array. After that, because of the rest parameter's presence, arr gets the rest of the values in the form of an array.
The rest element only works correctly as the last variable in the list. As in, you cannot use the rest parameter to catch a subarray that leaves out the last element of the original array.

2.16 Use Destructuring Assignment to Pass an Object as a Function's Parameters

In some cases, you can destructure the object in a function argument itself.
Consider the code below:
“‘js
const profileUpdate = (profileData) => {
const { name, age, nationality, location } = profileData;
// do something with these variables
}
“‘

This effectively destructures the object sent into the function. This can also be done in-place:
“‘js
const profileUpdate = ({ name, age, nationality, location }) => {
/* do something with these fields */
}
“‘

When profileData is passed to the above function, the values are destructured from the function parameter for use within the function.

2.17 Create Strings using Template Literals

A new feature of ES6 is the template literal. This is a special type of string that makes creating complex strings easier.
Template literals allow you to create multi-line strings and to use string interpolation features to create strings.
Consider the code below:
“‘js
const person = {
name: "Zodiac Hasbro",
age: 56
};
// Template literal with multi-line and string interpolation
const greeting = ‘Hello, my name is ${person.name}!
I am ${person.age} years old.‘;
console.log(greeting); // prints
// Hello, my name is Zodiac Hasbro!
// I am 56 years old.
“‘
A lot of things happened there.
Firstly, the example uses backticks (‘), not quotes (’ or "), to wrap the string.
Secondly, notice that the string is multi-line, both in the code and the output. This saves inserting \n within strings.
The ${variable} syntax used above is a placeholder. Basically, you won’t have to use concatenation with the + operator anymore. To add variables to strings, you just drop the variable in a template string and wrap it with ${ and }. Similarly, you can include other expressions in your string literal, for example ${a + b}.
This new way of creating strings gives you more flexibility to create robust strings.

2.18 Write Concise Object Literal Declarations Using Object Property Shorthand

ES6 adds some nice support for easily defining object literals.
Consider the following code:
“‘js
const getMousePosition = (x, y) => ({
x: x,
y: y
});
“‘
getMousePosition is a simple function that returns an object containing two properties.
ES6 provides the syntactic sugar to eliminate the redundancy of having to write x: x. You can simply write x once, and it will be converted tox: x (or something equivalent) under the hood.
Here is the same function from above rewritten to use this new syntax:
“‘js
const getMousePosition = (x, y) => ({ x, y });
“‘

2.19 Write Concise Declarative Functions with ES6

When defining functions within objects in ES5, we have to use the keyword function as follows:
“‘js
const person = {
name: "Taylor",

sayHello: function() {
return ‘Hello! My name is ${this.name}.‘;
}
};
“‘
With ES6, You can remove the function keyword and colon altogether when defining functions in objects. Here’s an example of this syntax:
“‘js
const person = {
name: "Taylor",
sayHello() {
return ‘Hello! My name is ${this.name}.‘;
}
};
“‘

2.20 Use class Syntax to Define a Constructor Function

ES6 provides a new syntax to create objects, using the class keyword.
It should be noted that the class syntax is just syntax, and not a full-fledged class-based implementation of an object-oriented paradigm, unlike in languages such as Java, Python, Ruby, etc.
In ES5, we usually define a constructor function and use the new keyword to instantiate an object.
“‘js
var SpaceShuttle = function(targetPlanet){
this.targetPlanet = targetPlanet;
}
var zeus = new SpaceShuttle(’Jupiter’);
“‘
The class syntax simply replaces the constructor function creation:
“‘js
class SpaceShuttle {
constructor(targetPlanet) {
this.targetPlanet = targetPlanet;
}
}
const zeus = new SpaceShuttle(’Jupiter’);
“‘
It should be noted that the class keyword declares a new function, to which a constructor is added. This constructor is invoked when new is called to create a new object.
Notes:
UpperCamelCase should be used by convention for ES6 class names, as in SpaceShuttle used above.
The constructor method is a special method for creating and initializing an object created with a class. You will learn more about it in the Object Oriented Programming section of the JavaScript Algorithms And Data Structures Certification.

2.21 Use getters and setters to Control Access to an Object

You can obtain values from an object and set the value of a property within an object.
These are classically called getters and setters.
Getter functions are meant to simply return (get) the value of an object’s private variable to the user without the user directly accessing the private variable.
Setter functions are meant to modify (set) the value of an object’s private variable based on the value passed

into the setter function. This change could involve calculations, or even overwriting the previous value completely.
“‘js
class Book {
constructor(author) {
this._author = author;
}
// getter
get writer() {
return this._author;
}
// setter
set writer(updatedAuthor) {
this._author = updatedAuthor;
}
}
const novel = new Book('anonymous');
console.log(novel.writer); // anonymous
novel.writer = 'newAuthor';
console.log(novel.writer); // newAuthor
“‘
Notice the syntax used to invoke the getter and setter. They do not even look like functions.
Getters and setters are important because they hide internal implementation details.
Note: It is convention to precede the name of a private variable with an underscore (_). However, the practice itself does not make a variable private.

2.22 Create a Module Script

JavaScript started with a small role to play on an otherwise mostly HTML web. Today, it's huge, and some websites are built almost entirely with JavaScript. In order to make JavaScript more modular, clean, and maintainable; ES6 introduced a way to easily share code among JavaScript files. This involves exporting parts of a file for use in one or more other files, and importing the parts you need, where you need them. In order to take advantage of this functionality, you need to create a script in your HTML document with a type of module. Here's an example:
“‘html
“‘
A script that uses this module type can now use the import and export features you will learn about in the upcoming challenges.

2.23 Use export to Share a Code Block

Imagine a file called math_functions.js that contains several functions related to mathematical operations. One of them is stored in a variable, add, that takes in two numbers and returns their sum. You want to use this function in several different JavaScript files. In order to share it with these other files, you first need to export it.
“‘js
export const add = (x, y) => {
return x + y;
}
“‘
The above is a common way to export a single function, but you can achieve the same thing like this:
“‘js

```
const add = (x, y) => {
return x + y;
}
export { add };
```

When you export a variable or function, you can import it in another file and use it without having to rewrite the code. You can export multiple things by repeating the first example for each thing you want to export, or by placing them all in the export statement of the second example, like this:

```js
export { add, subtract };
```

2.24 Reuse JavaScript Code Using import

import allows you to choose which parts of a file or module to load. In the previous lesson, the examples exported add from the math_functions.js file. Here's how you can import it to use in another file:

```js
import { add } from './math_functions.js';
```

Here, import will find add in math_functions.js, import just that function for you to use, and ignore the rest. The ./ tells the import to look for the math_functions.js file in the same folder as the current file. The relative file path (./) and file extension (.js) are required when using import in this way.
You can import more than one item from the file by adding them in the import statement like this:

```js
import { add, subtract } from './math_functions.js';
```

2.25 Use * to Import Everything from a File

Suppose you have a file and you wish to import all of its contents into the current file. This can be done with the import * as syntax. Here's an example where the contents of a file named math_functions.js are imported into a file in the same directory:

```js
import * as myMathModule from "./math_functions.js";
```

The above import statement will create an object called myMathModule. This is just a variable name, you can name it anything. The object will contain all of the exports from math_functions.js in it, so you can access the functions like you would any other object property. Here's how you can use the add and subtract functions that were imported:

```js
myMathModule.add(2,3);
myMathModule.subtract(5,3);
```

2.26 Create an Export Fallback with export default

In the export lesson, you learned about the syntax referred to as a named export. This allowed you to make multiple functions and variables available for use in other files.
There is another export syntax you need to know, known as export default. Usually you will use this syntax if only one value is being exported from a file. It is also used to create a fallback value for a file or module. Below are examples using export default:

```js
```

```
// named function
export default function add(x, y) {
return x + y;
}
// anonymous function
export default function(x, y) {
return x + y;
}
```

Since export default is used to declare a fallback value for a module or file, you can only have one value be a default export in each module or file. Additionally, you cannot use export default with var, let, or const

2.27 Import a Default Export

In the last challenge, you learned about export default and its uses. To import a default export, you need to use a different import syntax. In the following example, add is the default export of the math_functions.js file. Here is how to import it:

```js
import add from "./math_functions.js";
```

The syntax differs in one key place. The imported value, add, is not surrounded by curly braces ({}). add here is simply a variable name for whatever the default export of the math_functions.js file is. You can use any name here when importing a default.

2.28 Create a JavaScript Promise

A promise in JavaScript is exactly what it sounds like - you use it to make a promise to do something, usually asynchronously. When the task completes, you either fulfill your promise or fail to do so. Promise is a constructor function, so you need to use the new keyword to create one. It takes a function, as its argument, with two parameters - resolve and reject. These are methods used to determine the outcome of the promise. The syntax looks like this:

```js
const myPromise = new Promise((resolve, reject) => {
});
```

2.29 Complete a Promise with resolve and reject

A promise has three states: pending, fulfilled, and rejected. The promise you created in the last challenge is forever stuck in the pending state because you did not add a way to complete the promise. The resolve and reject parameters given to the promise argument are used to do this. resolve is used when you want your promise to succeed, and reject is used when you want it to fail. These are methods that take an argument, as seen below.

```js
const myPromise = new Promise((resolve, reject) => {
if(condition here) {
resolve("Promise was fulfilled");
} else {
reject("Promise was rejected");
}
});
```

The example above uses strings for the argument of these functions, but it can really be anything. Often, it might be an object, that you would use data from, to put on your website or elsewhere.

2.30 Handle a Fulfilled Promise with then

Promises are most useful when you have a process that takes an unknown amount of time in your code (i.e. something asynchronous), often a server request. When you make a server request it takes some amount of time, and after it completes you usually want to do something with the response from the server. This can be achieved by using the then method. The then method is executed immediately after your promise is fulfilled with resolve. Here's an example:
“‘js
myPromise.then(result => {
// do something with the result.
});
“‘
result comes from the argument given to the resolve method.

2.31 Handle a Rejected Promise with catch

catch is the method used when your promise has been rejected. It is executed immediately after a promise's reject method is called. Here's the syntax:
“‘js
myPromise.catch(error => {
// do something with the error.
});
“‘
error is the argument passed in to the reject method.

3 Regular Expressions

3.1 Using the Test Method

Regular expressions are used in programming languages to match parts of strings. You create patterns to help you do that matching.
If you want to find the word "the" in the string "The dog chased the cat", you could use the following regular expression: /the/. Notice that quote marks are not required within the regular expression.
JavaScript has multiple ways to use regexes. One way to test a regex is using the .test() method. The .test() method takes the regex, applies it to a string (which is placed inside the parentheses), and returns true or false if your pattern finds something or not.
“‘js
let testStr = "freeCodeCamp";
let testRegex = /Code/;
testRegex.test(testStr);
// Returns true
“‘

3.2 Match Literal Strings

In the last challenge, you searched for the word "Hello" using the regular expression /Hello/. That regex searched for a literal match of the string "Hello". Here's another example searching for a literal match of the string "Kevin":
“‘js
let testStr = "Hello, my name is Kevin.";
let testRegex = /Kevin/;
testRegex.test(testStr);
// Returns true
“‘
Any other forms of "Kevin" will not match. For example, the regex /Kevin/ will not match "kevin" or "KEVIN".
“‘js
let wrongRegex = /kevin/;
wrongRegex.test(testStr);
// Returns false
“‘
A future challenge will show how to match those other forms as well.

3.3 Match a Literal String with Different Possibilities

Using regexes like /coding/, you can look for the pattern "coding" in another string.
This is powerful to search single strings, but it's limited to only one pattern. You can search for multiple patterns using the alternation or OR operator: |.
This operator matches patterns either before or after it. For example, if you wanted to match "yes" or "no", the regex you want is /yes|no/.
You can also search for more than just two patterns. You can do this by adding more patterns with more OR operators separating them, like /yes|no|maybe/.

3.4 Ignore Case While Matching

Up until now, you've looked at regexes to do literal matches of strings. But sometimes, you might want to also match case differences.

Case (or sometimes letter case) is the difference between uppercase letters and lowercase letters. Examples of uppercase are "A", "B", and "C". Examples of lowercase are "a", "b", and "c".
You can match both cases using what is called a flag. There are other flags but here you'll focus on the flag that ignores case - the i flag. You can use it by appending it to the regex. An example of using this flag is /ignorecase/i. This regex can match the strings "ignorecase", "igNoreCase", and "IgnoreCase".

3.5 Extract Matches

So far, you have only been checking if a pattern exists or not within a string. You can also extract the actual matches you found with the .match() method.
To use the .match() method, apply the method on a string and pass in the regex inside the parentheses.
Here's an example:
“‘js
"Hello, World!".match(/Hello/);
// Returns ["Hello"]
let ourStr = "Regular expressions";
let ourRegex = /expressions/;
ourStr.match(ourRegex);
// Returns ["expressions"]
“‘
Note that the ‘.match‘ syntax is the "opposite" of the ‘.test‘ method you have been using thus far:
“‘js
'string'.match(/regex/);
/regex/.test('string');
“‘

3.6 Find More Than the First Match

So far, you have only been able to extract or search a pattern once.
“‘js
let testStr = "Repeat, Repeat, Repeat";
let ourRegex = /Repeat/;
testStr.match(ourRegex);
// Returns ["Repeat"]
“‘
To search or extract a pattern more than once, you can use the g flag.
“‘js
let repeatRegex = /Repeat/g;
testStr.match(repeatRegex);
// Returns ["Repeat", "Repeat", "Repeat"]
“‘

3.7 Match Anything with Wildcard Period

Sometimes you won't (or don't need to) know the exact characters in your patterns. Thinking of all words that match, say, a misspelling would take a long time. Luckily, you can save time using the wildcard character: .
The wildcard character . will match any one character. The wildcard is also called dot and period. You can use the wildcard character just like any other character in the regex. For example, if you wanted to match "hug", "huh", "hut", and "hum", you can use the regex /hu./ to match all four words.
“‘js
let humStr = "I'll hum a song";

```
let hugStr = "Bear hug";
let huRegex = /hu./;
huRegex.test(humStr); // Returns true
huRegex.test(hugStr); // Returns true
```

3.8 Match Single Character with Multiple Possibilities

You learned how to match literal patterns (/literal/) and wildcard character (/./). Those are the extremes of regular expressions, where one finds exact matches and the other matches everything. There are options that are a balance between the two extremes.
You can search for a literal pattern with some flexibility with character classes. Character classes allow you to define a group of characters you wish to match by placing them inside square ([and]) brackets.
For example, you want to match "bag", "big", and "bug" but not "bog". You can create the regex /b[aiu]g/ to do this. The [aiu] is the character class that will only match the characters "a", "i", or "u".
```js
let bigStr = "big";
let bagStr = "bag";
let bugStr = "bug";
let bogStr = "bog";
let bgRegex = /b[aiu]g/;
bigStr.match(bgRegex); // Returns ["big"]
bagStr.match(bgRegex); // Returns ["bag"]
bugStr.match(bgRegex); // Returns ["bug"]
bogStr.match(bgRegex); // Returns null
```

3.9 Match Letters of the Alphabet

You saw how you can use character sets to specify a group of characters to match, but that's a lot of typing when you need to match a large range of characters (for example, every letter in the alphabet). Fortunately, there is a built-in feature that makes this short and simple.
Inside a character set, you can define a range of characters to match using a hyphen character: -.
For example, to match lowercase letters a through e you would use [a-e].
```js
let catStr = "cat";
let batStr = "bat";
let matStr = "mat";
let bgRegex = /[a-e]at/;
catStr.match(bgRegex); // Returns ["cat"]
batStr.match(bgRegex); // Returns ["bat"]
matStr.match(bgRegex); // Returns null
```

3.10 Match Numbers and Letters of the Alphabet

Using the hyphen (-) to match a range of characters is not limited to letters. It also works to match a range of numbers.
For example, /[0-5]/ matches any number between 0 and 5, including the 0 and 5.
Also, it is possible to combine a range of letters and numbers in a single character set.
```js
let jennyStr = "Jenny8675309";
```

```
let myRegex = /[a-z0-9]/ig;
// matches all letters and numbers in jennyStr
jennyStr.match(myRegex);
```

3.11 Match Single Characters Not Specified

So far, you have created a set of characters that you want to match, but you could also create a set of characters that you do not want to match. These types of character sets are called negated character sets.
To create a negated character set, you place a caret character (^) after the opening bracket and before the characters you do not want to match.
For example, /[^aeiou]/gi matches all characters that are not a vowel. Note that characters like ., !, [, @, / and white space are matched - the negated vowel character set only excludes the vowel characters.

3.12 Match Characters that Occur One or More Times

Sometimes, you need to match a character (or group of characters) that appears one or more times in a row. This means it occurs at least once, and may be repeated.
You can use the + character to check if that is the case. Remember, the character or pattern has to be present consecutively. That is, the character has to repeat one after the other.
For example, /a+/g would find one match in "abc" and return ["a"]. Because of the +, it would also find a single match in "aabc" and return ["aa"].
If it were instead checking the string "abab", it would find two matches and return ["a", "a"] because the a characters are not in a row - there is a b between them. Finally, since there is no "a" in the string "bcd", it wouldn't find a match.

3.13 Match Characters that Occur Zero or More Times

The last challenge used the plus + sign to look for characters that occur one or more times. There's also an option that matches characters that occur zero or more times.
The character to do this is the asterisk or star: *.

```js
let soccerWord = "gooooooooal!";
let gPhrase = "gut feeling";
let oPhrase = "over the moon";
let goRegex = /go*/;
soccerWord.match(goRegex); // Returns ["gooooooooo"]
gPhrase.match(goRegex); // Returns ["g"]
oPhrase.match(goRegex); // Returns null
```

3.14 Find Characters with Lazy Matching

In regular expressions, a greedy match finds the longest possible part of a string that fits the regex pattern and returns it as a match. The alternative is called a lazy match, which finds the smallest possible part of the string that satisfies the regex pattern.
You can apply the regex /t[a-z]*i/ to the string "titanic". This regex is basically a pattern that starts with t, ends with i, and has some letters in between.
Regular expressions are by default greedy, so the match would return ["titani"]. It finds the largest sub-string possible to fit the pattern.
However, you can use the ? character to change it to lazy matching. "titanic" matched against the adjusted

regex of /t[a-z]*?i/ returns ["ti"].
NoteParsing HTML with regular expressions should be avoided, but pattern matching an HTML string with regular expressions is completely fine.

3.15 Find One or More Criminals in a Hunt

Time to pause and test your new regex writing skills. A group of criminals escaped from jail and ran away, but you don't know how many. However, you do know that they stay close together when they are around other people. You are responsible for finding all of the criminals at once.
Here's an example to review how to do this:
The regex /z+/ matches the letter z when it appears one or more times in a row. It would find matches in all of the following strings:
“‘js
"z"
"zzzzzz"
"ABCzzzz"
"zzzzABC"
"abczzzzzzzzzzzzzzzzzzzzzzzabc"
“‘
But it does not find matches in the following strings since there are no letter z characters:
“‘js
""
"ABC"
"abcabc"
“‘

3.16 Match Beginning String Patterns

Prior challenges showed that regular expressions can be used to look for a number of matches. They are also used to search for patterns in specific positions in strings.
In an earlier challenge, you used the caret character (^) inside a character set to create a negated character set in the form [^thingsThatWillNotBeMatched]. Outside of a character set, the caret is used to search for patterns at the beginning of strings.
“‘js
let firstString = "Ricky is first and can be found.";
let firstRegex = /^Ricky/;
firstRegex.test(firstString);
// Returns true
let notFirst = "You can't find Ricky now.";
firstRegex.test(notFirst);
// Returns false
“‘

3.17 Match Ending String Patterns

In the last challenge, you learned to use the caret character to search for patterns at the beginning of strings. There is also a way to search for patterns at the end of strings.
You can search the end of strings using the dollar sign character $ at the end of the regex.
“‘js
let theEnding = "This is a never ending story";
let storyRegex = /story$/;
storyRegex.test(theEnding);

```
// Returns true
let noEnding = "Sometimes a story will have to end";
storyRegex.test(noEnding);
// Returns false
```

3.18 Match All Letters and Numbers

Using character classes, you were able to search for all letters of the alphabet with [a-z]. This kind of character class is common enough that there is a shortcut for it, although it includes a few extra characters as well.
The closest character class in JavaScript to match the alphabet is \w. This shortcut is equal to [A-Za-z0-9_]. This character class matches upper and lowercase letters plus numbers. Note, this character class also includes the underscore character (_).

```js
let longHand = /[A-Za-z0-9_]+/;
let shortHand = /\w+/;
let numbers = "42";
let varNames = "important_var";
longHand.test(numbers); // Returns true
shortHand.test(numbers); // Returns true
longHand.test(varNames); // Returns true
shortHand.test(varNames); // Returns true
```

These shortcut character classes are also known as shorthand character classes.

3.19 Match Everything But Letters and Numbers

You've learned that you can use a shortcut to match alphanumerics [A-Za-z0-9_] using \w. A natural pattern you might want to search for is the opposite of alphanumerics.
You can search for the opposite of the \w with \W. Note, the opposite pattern uses a capital letter. This shortcut is the same as [^A-Za-z0-9_].

```js
let shortHand = /\W/;
let numbers = "42%";
let sentence = "Coding!";
numbers.match(shortHand); // Returns ["%"]
sentence.match(shortHand); // Returns ["!"]
```

3.20 Match All Numbers

You've learned shortcuts for common string patterns like alphanumerics. Another common pattern is looking for just digits or numbers.
The shortcut to look for digit characters is \d, with a lowercase d. This is equal to the character class [0-9], which looks for a single character of any number between zero and nine.

3.21 Match All Non-Numbers

The last challenge showed how to search for digits using the shortcut \d with a lowercase d. You can also search for non-digits using a similar shortcut that uses an uppercase D instead.

The shortcut to look for non-digit characters is \D. This is equal to the character class [^0-9], which looks for a single character that is not a number between zero and nine.

3.22 Restrict Possible Usernames

Usernames are used everywhere on the internet. They are what give users a unique identity on their favorite sites.
You need to check all the usernames in a database. Here are some simple rules that users have to follow when creating their username.
1) Usernames can only use alpha-numeric characters.
2) The only numbers in the username have to be at the end. There can be zero or more of them at the end. Username cannot start with the number.
3) Username letters can be lowercase and uppercase.
4) Usernames have to be at least two characters long. A two-character username can only use alphabet letters as characters.

3.23 Match Whitespace

The challenges so far have covered matching letters of the alphabet and numbers. You can also match the whitespace or spaces between letters.
You can search for whitespace using \s, which is a lowercase s. This pattern not only matches whitespace, but also carriage return, tab, form feed, and new line characters. You can think of it as similar to the character class [\r\t\f\n\v].
“‘js
let whiteSpace = "Whitespace. Whitespace everywhere!"
let spaceRegex = /\s/g;
whiteSpace.match(spaceRegex);
// Returns [" ", " "]
“‘

3.24 Match Non-Whitespace Characters

You learned about searching for whitespace using \s, with a lowercase s. You can also search for everything except whitespace.
Search for non-whitespace using \S, which is an uppercase s. This pattern will not match whitespace, carriage return, tab, form feed, and new line characters. You can think of it being similar to the character class [^ \r\t\f\n\v].
“‘js
let whiteSpace = "Whitespace. Whitespace everywhere!"
let nonSpaceRegex = /\S/g;
whiteSpace.match(nonSpaceRegex).length; // Returns 32
“‘

3.25 Specify Upper and Lower Number of Matches

Recall that you use the plus sign + to look for one or more characters and the asterisk * to look for zero or more characters. These are convenient but sometimes you want to match a certain range of patterns.
You can specify the lower and upper number of patterns with quantity specifiers. Quantity specifiers are used with curly brackets ({ and }). You put two numbers between the curly brackets - for the lower and upper number of patterns.
For example, to match only the letter a appearing between 3 and 5 times in the string "ah", your regex would

be /a{3,5}h/.
“‘js
let A4 = "aaaah";
let A2 = "aah";
let multipleA = /a{3,5}h/;
multipleA.test(A4); // Returns true
multipleA.test(A2); // Returns false
“‘

3.26 Specify Only the Lower Number of Matches

You can specify the lower and upper number of patterns with quantity specifiers using curly brackets. Sometimes you only want to specify the lower number of patterns with no upper limit.
To only specify the lower number of patterns, keep the first number followed by a comma.
For example, to match only the string "hah" with the letter a appearing at least 3 times, your regex would be /ha{3,}h/.
“‘js
let A4 = "haaaah";
let A2 = "haah";
let A100 = "h" + "a".repeat(100) + "h";
let multipleA = /ha{3,}h/;
multipleA.test(A4); // Returns true
multipleA.test(A2); // Returns false
multipleA.test(A100); // Returns true
“‘

3.27 Specify Exact Number of Matches

You can specify the lower and upper number of patterns with quantity specifiers using curly brackets. Sometimes you only want a specific number of matches.
To specify a certain number of patterns, just have that one number between the curly brackets.
For example, to match only the word "hah" with the letter a 3 times, your regex would be /ha{3}h/.
“‘js
let A4 = "haaaah";
let A3 = "haaah";
let A100 = "h" + "a".repeat(100) + "h";
let multipleHA = /ha{3}h/;
multipleHA.test(A4); // Returns false
multipleHA.test(A3); // Returns true
multipleHA.test(A100); // Returns false
“‘

3.28 Check for All or None

Sometimes the patterns you want to search for may have parts of it that may or may not exist. However, it may be important to check for them nonetheless.
You can specify the possible existence of an element with a question mark, ?. This checks for zero or one of the preceding element. You can think of this symbol as saying the previous element is optional.
For example, there are slight differences in American and British English and you can use the question mark to match both spellings.
“‘js
let american = "color";

```
let british = "colour";
let rainbowRegex= /colou?r/;
rainbowRegex.test(american); // Returns true
rainbowRegex.test(british); // Returns true
```

3.29 Positive and Negative Lookahead

Lookaheads are patterns that tell JavaScript to look-ahead in your string to check for patterns further along. This can be useful when you want to search for multiple patterns over the same string.
There are two kinds of lookaheads: positive lookahead and negative lookahead.
A positive lookahead will look to make sure the element in the search pattern is there, but won't actually match it. A positive lookahead is used as (?=...) where the ... is the required part that is not matched.
On the other hand, a negative lookahead will look to make sure the element in the search pattern is not there. A negative lookahead is used as (?!...) where the ... is the pattern that you do not want to be there. The rest of the pattern is returned if the negative lookahead part is not present.
Lookaheads are a bit confusing but some examples will help.

```js
let quit = "qu";
let noquit = "qt";
let quRegex= /q(?=u)/;
let qRegex = /q(?!u)/;
quit.match(quRegex); // Returns ["q"]
noquit.match(qRegex); // Returns ["q"]
```

A more practical use of lookaheads is to check two or more patterns in one string. Here is a (naively) simple password checker that looks for between 3 and 6 characters and at least one number:

```js
let password = "abc123";
let checkPass = /(?=\w{3,6})(?=\D*\d)/;
checkPass.test(password); // Returns true
```

3.30 Check For Mixed Grouping of Characters

Sometimes we want to check for groups of characters using a Regular Expression and to achieve that we use parentheses ().
If you want to find either Penguin or Pumpkin in a string, you can use the following Regular Expression: /P(engu|umpk)in/g
Then check whether the desired string groups are in the test string by using the test() method.

```js
let testStr = "Pumpkin";
let testRegex = /P(engu|umpk)in/;
testRegex.test(testStr);
// Returns true
```

3.31 Reuse Patterns Using Capture Groups

Some patterns you search for will occur multiple times in a string. It is wasteful to manually repeat that regex. There is a better way to specify when you have multiple repeat substrings in your string.
You can search for repeat substrings using capture groups. Parentheses, (and), are used to find repeat

substrings. You put the regex of the pattern that will repeat in between the parentheses.
To specify where that repeat string will appear, you use a backslash (\) and then a number. This number starts at 1 and increases with each additional capture group you use. An example would be \1 to match the first group.
The example below matches any word that occurs twice separated by a space:
“‘js
let repeatStr = "regex regex";
let repeatRegex = /(\w+)\s\1/;
repeatRegex.test(repeatStr); // Returns true
repeatStr.match(repeatRegex); // Returns ["regex regex", "regex"]
“‘
Using the .match() method on a string will return an array with the string it matches, along with its capture group.

3.32 Use Capture Groups to Search and Replace

Searching is useful. However, you can make searching even more powerful when it also changes (or replaces) the text you match.
You can search and replace text in a string using .replace() on a string. The inputs for .replace() is first the regex pattern you want to search for. The second parameter is the string to replace the match or a function to do something.
“‘js
let wrongText = "The sky is silver.";
let silverRegex = /silver/;
wrongText.replace(silverRegex, "blue");
// Returns "The sky is blue."
“‘
You can also access capture groups in the replacement string with dollar signs ($).
“‘js
"Code Camp".replace(/(\w+)\s(\w+)/, ’$2 $1’);
// Returns "Camp Code"
“‘

3.33 Remove Whitespace from Start and End

Sometimes whitespace characters around strings are not wanted but are there. Typical processing of strings is to remove the whitespace at the start and end of it.

4 Debugging

4.1 Use the JavaScript Console to Check the Value of a Variable

Both Chrome and Firefox have excellent JavaScript consoles, also known as DevTools, for debugging your JavaScript.
You can find Developer tools in your Chrome's menu or Web Console in Firefox's menu. If you're using a different browser, or a mobile phone, we strongly recommend switching to desktop Firefox or Chrome.
The console.log() method, which "prints" the output of what's within its parentheses to the console, will likely be the most helpful debugging tool. Placing it at strategic points in your code can show you the intermediate values of variables. It's good practice to have an idea of what the output should be before looking at what it is. Having check points to see the status of your calculations throughout your code will help narrow down where the problem is.
Here's an example to print 'Hello world!' to the console:
console.log('Hello world!');

4.2 Understanding the Differences between the freeCodeCamp and Browser Console

You may have noticed that some freeCodeCamp JavaScript challenges include their own console. This console behaves a little differently than the browser console you used in the last challenge.
The following challenge is meant to highlight the main difference between the freeCodeCamp console and your browser console.
When you run ordinary JavaScript, the browser's console will display your console.log() statements the exact number of times it is called.
The freeCodeCamp console will print your console.log() statements a short time after the editor detects a change in the script, as well as during testing.
The freeCodeCamp console is cleared before the tests are run and, to avoid spam, only prints the logs during the first test (see the note below for exceptions).
If you would like to see every log for every test, run the tests, and open the browser console. If you prefer to use the browser console, and want it to mimic the freeCodeCamp console, place console.clear() before any other console calls, to clear the browser console.
Note: 'console.log's inside functions are printed to the freeCodeCamp console whenever those functions are called, this can help debugging functions that are called during testing.

4.3 Use typeof to Check the Type of a Variable

You can use typeof to check the data structure, or type, of a variable. This is useful in debugging when working with multiple data types. If you think you're adding two numbers, but one is actually a string, the results can be unexpected. Type errors can lurk in calculations or function calls. Be careful especially when you're accessing and working with external data in the form of a JavaScript Object Notation (JSON) object.
Here are some examples using typeof:
"'js
console.log(typeof ""); // outputs "string"
console.log(typeof 0); // outputs "number"
console.log(typeof []); // outputs "object"
console.log(typeof {}); // outputs "object"
"'
JavaScript recognizes six primitive (immutable) data types: Boolean, Null, Undefined, Number, String, and Symbol (new with ES6) and one type for mutable items: Object. Note that in JavaScript, arrays are technically a type of object.

4.4 Catch Misspelled Variable and Function Names

The console.log() and typeof methods are the two primary ways to check intermediate values and types of program output. Now it's time to get into the common forms that bugs take. One syntax-level issue that fast typers can commiserate with is the humble spelling error.
Transposed, missing, or mis-capitalized characters in a variable or function name will have the browser looking for an object that doesn't exist - and complain in the form of a reference error. JavaScript variable and function names are case-sensitive.

4.5 Catch Unclosed Parentheses, Brackets, Braces and Quotes

Another syntax error to be aware of is that all opening parentheses, brackets, curly braces, and quotes have a closing pair. Forgetting a piece tends to happen when you're editing existing code and inserting items with one of the pair types. Also, take care when nesting code blocks into others, such as adding a callback function as an argument to a method.
One way to avoid this mistake is as soon as the opening character is typed, immediately include the closing match, then move the cursor back between them and continue coding. Fortunately, most modern code editors generate the second half of the pair automatically.

4.6 Catch Mixed Usage of Single and Double Quotes

JavaScript allows the use of both single (') and double (") quotes to declare a string. Deciding which one to use generally comes down to personal preference, with some exceptions.
Having two choices is great when a string has contractions or another piece of text that's in quotes. Just be careful that you don't close the string too early, which causes a syntax error.
Here are some examples of mixing quotes:
“‘js
// These are correct:
const grouchoContraction = "I've had a perfectly wonderful evening, but this wasn't it.";
const quoteInString = "Groucho Marx once said 'Quote me as saying I was mis-quoted.'";
// This is incorrect:
const uhOhGroucho = 'I've had a perfectly wonderful evening, but this wasn't it.';
“‘
Of course, it is okay to use only one style of quotes. You can escape the quotes inside the string by using the backslash (\\) escape character:
“‘js
// Correct use of same quotes:
const allSameQuotes = 'I\'ve had a perfectly wonderful evening, but this wasn\'t it.';
“‘

4.7 Catch Use of Assignment Operator Instead of Equality Operator

Branching programs, i.e. ones that do different things if certain conditions are met, rely on if, else if, and else statements in JavaScript. The condition sometimes takes the form of testing whether a result is equal to a value.
This logic is spoken (in English, at least) as "if x equals y, then ..." which can literally translate into code using the =, or assignment operator. This leads to unexpected control flow in your program.
As covered in previous challenges, the assignment operator (=) in JavaScript assigns a value to a variable name. And the == and === operators check for equality (the triple === tests for strict equality, meaning both value and type are the same).
The code below assigns x to be 2, which evaluates as true. Almost every value on its own in JavaScript evaluates to true, except what are known as the "falsy" values: false, 0, "" (an empty string), NaN, undefined,

and null.
“‘js
let x = 1;
let y = 2;
if (x = y) {
// this code block will run for any value of y (unless y were originally set as a falsy)
} else {
// this code block is what should run (but won't) in this example
}
“‘

4.8 Catch Missing Open and Closing Parenthesis After a Function Call

When a function or method doesn't take any arguments, you may forget to include the (empty) opening and closing parentheses when calling it. Often times the result of a function call is saved in a variable for other use in your code. This error can be detected by logging variable values (or their types) to the console and seeing that one is set to a function reference, instead of the expected value the function returns.
The variables in the following example are different:
“‘js
function myFunction() {
return "You rock!";
}
let varOne = myFunction; // set to equal a function
let varTwo = myFunction(); // set to equal the string "You rock!"
“‘

4.9 Catch Arguments Passed in the Wrong Order When Calling a Function

Continuing the discussion on calling functions, the next bug to watch out for is when a function's arguments are supplied in the incorrect order. If the arguments are different types, such as a function expecting an array and an integer, this will likely throw a runtime error. If the arguments are the same type (all integers, for example), then the logic of the code won't make sense. Make sure to supply all required arguments, in the proper order to avoid these issues.

4.10 Catch Off By One Errors When Using Indexing

Off by one errors (sometimes called OBOE) crop up when you're trying to target a specific index of a string or array (to slice or access a segment), or when looping over the indices of them. JavaScript indexing starts at zero, not one, which means the last index is always one less than the length of the item. If you try to access an index equal to the length, the program may throw an "index out of range" reference error or print undefined.
When you use string or array methods that take index ranges as arguments, it helps to read the documentation and understand if they are inclusive (the item at the given index is part of what's returned) or not. Here are some examples of off by one errors:
“‘js
let alphabet = "abcdefghijklmnopqrstuvwxyz";
let len = alphabet.length;
for (let i = 0; i <= len; i++) {
// loops one too many times at the end
console.log(alphabet[i]);
}
for (let j = 1; j < len; j++) {

```
// loops one too few times and misses the first character at index 0
console.log(alphabet[j]);
}
for (let k = 0; k < len; k++) {
// Goldilocks approves - this is just right
console.log(alphabet[k]);
}
```

4.11 Use Caution When Reinitializing Variables Inside a Loop

Sometimes it's necessary to save information, increment counters, or re-set variables within a loop. A potential issue is when variables either should be reinitialized, and aren't, or vice versa. This is particularly dangerous if you accidentally reset the variable being used for the terminal condition, causing an infinite loop.

Printing variable values with each cycle of your loop by using console.log() can uncover buggy behavior related to resetting, or failing to reset a variable.

4.12 Prevent Infinite Loops with a Valid Terminal Condition

The final topic is the dreaded infinite loop. Loops are great tools when you need your program to run a code block a certain number of times or until a condition is met, but they need a terminal condition that ends the looping. Infinite loops are likely to freeze or crash the browser, and cause general program execution mayhem, which no one wants.

There was an example of an infinite loop in the introduction to this section - it had no terminal condition to break out of the while loop inside loopy(). Do NOT call this function!

```js
function loopy() {
while(true) {
console.log("Hello, world!");
}
}
```

It's the programmer's job to ensure that the terminal condition, which tells the program when to break out of the loop code, is eventually reached. One error is incrementing or decrementing a counter variable in the wrong direction from the terminal condition. Another one is accidentally resetting a counter or index variable within the loop code, instead of incrementing or decrementing it.

5 Basic Data Structures

5.1 Use an Array to Store a Collection of Data

The below is an example of the simplest implementation of an array data structure. This is known as a one-dimensional array, meaning it only has one level, or that it does not have any other arrays nested within it. Notice it contains booleans, strings, and numbers, among other valid JavaScript data types:

```js
let simpleArray = ['one', 2, 'three', true, false, undefined, null];
console.log(simpleArray.length);
// logs 7
```

All arrays have a length property, which as shown above, can be very easily accessed with the syntax Array.length.

A more complex implementation of an array can be seen below. This is known as a multi-dimensional array, or an array that contains other arrays. Notice that this array also contains JavaScript objects, which we will examine very closely in our next section, but for now, all you need to know is that arrays are also capable of storing complex objects.

```js
let complexArray = [
[
{
one: 1,
two: 2
},
{
three: 3,
four: 4
}
],
[
{
a: "a",
b: "b"
},
{
c: "c",
d: "d"
}
]
];
```

5.2 Access an Array's Contents Using Bracket Notation

The fundamental feature of any data structure is, of course, the ability to not only store data, but to be able to retrieve that data on command. So, now that we've learned how to create an array, let's begin to think about how we can access that array's information.

When we define a simple array as seen below, there are 3 items in it:

```js
let ourArray = ["a", "b", "c"];
```

In an array, each array item has an index. This index doubles as the position of that item in the array, and how you reference it. However, it is important to note, that JavaScript arrays are zero-indexed, meaning

that the first element of an array is actually at the zeroth position, not the first.
In order to retrieve an element from an array we can enclose an index in brackets and append it to the end of an array, or more commonly, to a variable which references an array object. This is known as bracket notation.
For example, if we want to retrieve the "a" from ourArray and assign it to a variable, we can do so with the following code:
“‘js
let ourVariable = ourArray[0];
// ourVariable equals "a"
“‘
In addition to accessing the value associated with an index, you can also set an index to a value using the same notation:
“‘js
ourArray[1] = "not b anymore";
// ourArray now equals ["a", "not b anymore", "c"];
“‘
Using bracket notation, we have now reset the item at index 1 from "b", to "not b anymore".

5.3 Add Items to an Array with push() and unshift()

An array's length, like the data types it can contain, is not fixed. Arrays can be defined with a length of any number of elements, and elements can be added or removed over time; in other words, arrays are mutable. In this challenge, we will look at two methods with which we can programmatically modify an array: Array.push() and Array.unshift().
Both methods take one or more elements as parameters and add those elements to the array the method is being called on; the push() method adds elements to the end of an array, and unshift() adds elements to the beginning. Consider the following:
“‘js
let twentyThree = 'XXIII';
let romanNumerals = ['XXI', 'XXII'];
romanNumerals.unshift('XIX', 'XX');
// now equals ['XIX', 'XX', 'XXI', 'XXII']
romanNumerals.push(twentyThree);
// now equals ['XIX', 'XX', 'XXI', 'XXII', 'XXIII']Notice that we can also pass variables, which allows us even greater flexibility in dynamically modifying our array's data.
“‘

5.4 Remove Items from an Array with pop() and shift()

Both push() and unshift() have corresponding methods that are nearly functional opposites: pop() and shift(). As you may have guessed by now, instead of adding, pop() removes an element from the end of an array, while shift() removes an element from the beginning. The key difference between pop() and shift() and their cousins push() and unshift(), is that neither method takes parameters, and each only allows an array to be modified by a single element at a time.
Let's take a look:
“‘js
let greetings = ['whats up?', 'hello', 'see ya!'];
greetings.pop();
// now equals ['whats up?', 'hello']
greetings.shift();
// now equals ['hello']
“‘

We can also return the value of the removed element with either method like this:
“‘js
let popped = greetings.pop();
// returns ’hello’
// greetings now equals []
“‘

5.5 Remove Items Using splice()

Ok, so we’ve learned how to remove elements from the beginning and end of arrays using shift() and pop(), but what if we want to remove an element from somewhere in the middle? Or remove more than one element at once? Well, that’s where splice() comes in. splice() allows us to do just that: remove any number of consecutive elements from anywhere in an array.
splice() can take up to 3 parameters, but for now, we’ll focus on just the first 2. The first two parameters of splice() are integers which represent indexes, or positions, of the array that splice() is being called upon. And remember, arrays are zero-indexed, so to indicate the first element of an array, we would use 0. splice()’s first parameter represents the index on the array from which to begin removing elements, while the second parameter indicates the number of elements to delete. For example:
“‘js
let array = [’today’, ’was’, ’not’, ’so’, ’great’];
array.splice(2, 2);
// remove 2 elements beginning with the 3rd element
// array now equals [’today’, ’was’, ’great’]
“‘
splice() not only modifies the array it’s being called on, but it also returns a new array containing the value of the removed elements:
“‘js
let array = [’I’, ’am’, ’feeling’, ’really’, ’happy’];
let newArray = array.splice(3, 2);
// newArray equals [’really’, ’happy’]
“‘

5.6 Add Items Using splice()

Remember in the last challenge we mentioned that splice() can take up to three parameters? Well, you can use the third parameter, comprised of one or more element(s), to add to the array. This can be incredibly useful for quickly switching out an element, or a set of elements, for another.
“‘js
const numbers = [10, 11, 12, 12, 15];
const startIndex = 3;
const amountToDelete = 1;
numbers.splice(startIndex, amountToDelete, 13, 14);
// the second entry of 12 is removed, and we add 13 and 14 at the same index
console.log(numbers);
// returns [10, 11, 12, 13, 14, 15]
“‘
Here we begin with an array of numbers. We then pass the following to splice(). The index at which to begin deleting elements (3), the number of elements to be deleted (1), and the elements (13, 14) to be inserted at that same index. Note that there can be any number of elements (separated by commas) following amountToDelete, each of which gets inserted.

5.7 Copy Array Items Using slice()

The next method we will cover is slice(). Rather than modifying an array, slice() copies or extracts a given number of elements to a new array, leaving the array it is called upon untouched. slice() takes only 2 parameters — the first is the index at which to begin extraction, and the second is the index at which to stop extraction (extraction will occur up to, but not including the element at this index). Consider this:

```
“‘js
let weatherConditions = [’rain’, ’snow’, ’sleet’, ’hail’, ’clear’];
let todaysWeather = weatherConditions.slice(1, 3);
// todaysWeather equals [’snow’, ’sleet’];
// weatherConditions still equals [’rain’, ’snow’, ’sleet’, ’hail’, ’clear’]
“‘
```

In effect, we have created a new array by extracting elements from an existing array.

5.8 Copy an Array with the Spread Operator

While slice() allows us to be selective about what elements of an array to copy, among several other useful tasks, ES6’s new spread operator allows us to easily copy all of an array’s elements, in order, with a simple and highly readable syntax. The spread syntax simply looks like this: ...
In practice, we can use the spread operator to copy an array like so:

```
“‘js
let thisArray = [true, true, undefined, false, null];
let thatArray = [...thisArray];
// thatArray equals [true, true, undefined, false, null]
// thisArray remains unchanged and thatArray contains the same elements as thisArray
“‘
```

5.9 Combine Arrays with the Spread Operator

Another huge advantage of the spread operator, is the ability to combine arrays, or to insert all the elements of one array into another, at any index. With more traditional syntaxes, we can concatenate arrays, but this only allows us to combine arrays at the end of one, and at the start of another. Spread syntax makes the following operation extremely simple:

```
“‘js
let thisArray = [’sage’, ’rosemary’, ’parsley’, ’thyme’];
let thatArray = [’basil’, ’cilantro’, ...thisArray, ’coriander’];
// thatArray now equals [’basil’, ’cilantro’, ’sage’, ’rosemary’, ’parsley’, ’thyme’, ’coriander’]
“‘
```

Using spread syntax, we have just achieved an operation that would have been more complex and more verbose had we used traditional methods.

5.10 Check For The Presence of an Element With indexOf()

Since arrays can be changed, or mutated, at any time, there’s no guarantee about where a particular piece of data will be on a given array, or if that element even still exists. Luckily, JavaScript provides us with another built-in method, indexOf(), that allows us to quickly and easily check for the presence of an element on an array. indexOf() takes an element as a parameter, and when called, it returns the position, or index, of that element, or -1 if the element does not exist on the array.
For example:

```
“‘js
let fruits = [’apples’, ’pears’, ’oranges’, ’peaches’, ’pears’];
fruits.indexOf(’dates’); // returns -1
```

fruits.indexOf('oranges'); // returns 2
fruits.indexOf('pears'); // returns 1, the first index at which the element exists
‘‘‘

5.11 Iterate Through All an Array's Items Using For Loops

Sometimes when working with arrays, it is very handy to be able to iterate through each item to find one or more elements that we might need, or to manipulate an array based on which data items meet a certain set of criteria. JavaScript offers several built in methods that each iterate over arrays in slightly different ways to achieve different results (such as every(), forEach(), map(), etc.), however the technique which is most flexible and offers us the greatest amount of control is a simple for loop.
Consider the following:
“‘js
function greaterThanTen(arr) {
let newArr = [];
for (let i = 0; i < arr.length; i++) {
if (arr[i] > 10) {
newArr.push(arr[i]);
}
}
return newArr;
}
greaterThanTen([2, 12, 8, 14, 80, 0, 1]);
// returns [12, 14, 80]
‘‘‘
Using a for loop, this function iterates through and accesses each element of the array, and subjects it to a simple test that we have created. In this way, we have easily and programmatically determined which data items are greater than 10, and returned a new array containing those items.

5.12 Create complex multi-dimensional arrays

Awesome! You have just learned a ton about arrays! This has been a fairly high level overview, and there is plenty more to learn about working with arrays, much of which you will see in later sections. But before moving on to looking at Objects, lets take one more look, and see how arrays can become a bit more complex than what we have seen in previous challenges.
One of the most powerful features when thinking of arrays as data structures, is that arrays can contain, or even be completely made up of other arrays. We have seen arrays that contain arrays in previous challenges, but fairly simple ones. However, arrays can contain an infinite depth of arrays that can contain other arrays, each with their own arbitrary levels of depth, and so on. In this way, an array can very quickly become very complex data structure, known as a multi-dimensional, or nested array. Consider the following example:
“‘js
let nestedArray = [// top, or first level - the outer most array
['deep'], // an array within an array, 2 levels of depth
[
['deeper'], ['deeper'] // 2 arrays nested 3 levels deep
],
[
[
['deepest'], ['deepest'] // 2 arrays nested 4 levels deep
],
[
[

['deepest-est?'] // an array nested 5 levels deep
]
]
]
];
‘‘‘
While this example may seem convoluted, this level of complexity is not unheard of, or even unusual, when dealing with large amounts of data.
However, we can still very easily access the deepest levels of an array this complex with bracket notation:
“‘js
console.log(nestedArray[2][1][0][0][0]);
// logs: deepest-est?
‘‘‘
And now that we know where that piece of data is, we can reset it if we need to:
“‘js
nestedArray[2][1][0][0][0] = 'deeper still';
console.log(nestedArray[2][1][0][0][0]);
// now logs: deeper still
‘‘‘

5.13 Add Key-Value Pairs to JavaScript Objects

At their most basic, objects are just collections of key-value pairs. In other words, they are pieces of data (values) mapped to unique identifiers called properties (keys). Take a look at an example:
“‘js
const tekkenCharacter = {
player: 'Hwoarang',
fightingStyle: 'Tae Kwon Doe',
human: true
};
‘‘‘
The above code defines a Tekken video game character object called tekkenCharacter. It has three properties, each of which map to a specific value. If you want to add an additional property, such as "origin", it can be done by assigning origin to the object:
“‘js
tekkenCharacter.origin = 'South Korea';
‘‘‘
This uses dot notation. If you were to observe the tekkenCharacter object, it will now include the origin property. Hwoarang also had distinct orange hair. You can add this property with bracket notation by doing:
“‘js
tekkenCharacter['hair color'] = 'dyed orange';
‘‘‘
Bracket notation is required if your property has a space in it or if you want to use a variable to name the property. In the above case, the property is enclosed in quotes to denote it as a string and will be added exactly as shown. Without quotes, it will be evaluated as a variable and the name of the property will be whatever value the variable is. Here's an example with a variable:
“‘js
const eyes = 'eye color';
tekkenCharacter[eyes] = 'brown';
‘‘‘
After adding all the examples, the object will look like this:
“‘js
{

player: 'Hwoarang',
fightingStyle: 'Tae Kwon Doe',
human: true,
origin: 'South Korea',
'hair color': 'dyed orange',
'eye color': 'brown'
};
```

## 5.14 Modify an Object Nested Within an Object

Now let's take a look at a slightly more complex object. Object properties can be nested to an arbitrary depth, and their values can be any type of data supported by JavaScript, including arrays and even other objects. Consider the following:

```js
let nestedObject = {
id: 28802695164,
date: 'December 31, 2016',
data: {
totalUsers: 99,
online: 80,
onlineStatus: {
active: 67,
away: 13,
busy: 8
}
}
};
```

nestedObject has three properties: id (value is a number), date (value is a string), and data (value is an object with its nested structure). While structures can quickly become complex, we can still use the same notations to access the information we need. To assign the value 10 to the busy property of the nested onlineStatus object, we use dot notation to reference the property:

```js
nestedObject.data.onlineStatus.busy = 10;
```

5.15 Access Property Names with Bracket Notation

In the first object challenge we mentioned the use of bracket notation as a way to access property values using the evaluation of a variable. For instance, imagine that our foods object is being used in a program for a supermarket cash register. We have some function that sets the selectedFood and we want to check our foods object for the presence of that food. This might look like:

```js
let selectedFood = getCurrentFood(scannedItem);
let inventory = foods[selectedFood];
```

This code will evaluate the value stored in the selectedFood variable and return the value of that key in the foods object, or undefined if it is not present. Bracket notation is very useful because sometimes object properties are not known before runtime or we need to access them in a more dynamic way.
```

5.16 Use the delete Keyword to Remove Object Properties

Now you know what objects are and their basic features and advantages. In short, they are key-value stores which provide a flexible, intuitive way to structure data, and, they provide very fast lookup time. Throughout the rest of these challenges, we will describe several common operations you can perform on objects so you can become comfortable applying these useful data structures in your programs.
In earlier challenges, we have both added to and modified an object's key-value pairs. Here we will see how we can remove a key-value pair from an object.
Let's revisit our foods object example one last time. If we wanted to remove the apples key, we can remove it by using the delete keyword like this:
“‘js
delete foods.apples;
“‘

5.17 Check if an Object has a Property

Now we can add, modify, and remove keys from objects. But what if we just wanted to know if an object has a specific property? JavaScript provides us with two different ways to do this. One uses the hasOwnProperty() method and the other uses the in keyword. If we have an object users with a property of Alan, we could check for its presence in either of the following ways:
“‘js
users.hasOwnProperty('Alan');
'Alan' in users;
// both return true
“‘

5.18 Iterate Through the Keys of an Object with a for...in Statement

Sometimes you may need to iterate through all the keys within an object. This requires a specific syntax in JavaScript called a for...in statement. For our users object, this could look like:
“‘js
for (let user in users) {
console.log(user);
}
// logs:
Alan
Jeff
Sarah
Ryan
“‘
In this statement, we defined a variable user, and as you can see, this variable was reset during each iteration to each of the object's keys as the statement looped through the object, resulting in each user's name being printed to the console.
NOTE: Objects do not maintain an ordering to stored keys like arrays do; thus a key's position on an object, or the relative order in which it appears, is irrelevant when referencing or accessing that key.

5.19 Generate an Array of All Object Keys with Object.keys()

We can also generate an array which contains all the keys stored in an object using the Object.keys() method and passing in an object as the argument. This will return an array with strings representing each property in the object. Again, there will be no specific order to the entries in the array.

5.20 Modify an Array Stored in an Object

Now you've seen all the basic operations for JavaScript objects. You can add, modify, and remove key-value pairs, check if keys exist, and iterate over all the keys in an object. As you continue learning JavaScript you will see even more versatile applications of objects. Additionally, the Data Structures lessons located in the Coding Interview Prep section of the curriculum also cover the ES6 Map and Set objects, both of which are similar to ordinary objects but provide some additional features. Now that you've learned the basics of arrays and objects, you're fully prepared to begin tackling more complex problems using JavaScript!

6 Basic Algorithm Scripting

6.1 Convert Celsius to Fahrenheit

The algorithm to convert from Celsius to Fahrenheit is the temperature in Celsius times 9/5, plus 32.
You are given a variable celsius representing a temperature in Celsius. Use the variable fahrenheit already defined and assign it the Fahrenheit temperature equivalent to the given Celsius temperature. Use the algorithm mentioned above to help convert the Celsius temperature to Fahrenheit.

6.2 Reverse a String

Reverse the provided string.
You may need to turn the string into an array before you can reverse it.
Your result must be a string.

6.3 Factorialize a Number

Return the factorial of the provided integer.
If the integer is represented with the letter n, a factorial is the product of all positive integers less than or equal to n.
Factorials are often represented with the shorthand notation n!
For example: 5! = 1 * 2 * 3 * 4 * 5 = 120
Only integers greater than or equal to zero will be supplied to the function.

6.4 Find the Longest Word in a String

Return the length of the longest word in the provided sentence.
Your response should be a number.

6.5 Return Largest Numbers in Arrays

Return an array consisting of the largest number from each provided sub-array. For simplicity, the provided array will contain exactly 4 sub-arrays.
Remember, you can iterate through an array with a simple for loop, and access each member with array syntax arr[i].

6.6 Confirm the Ending

Check if a string (first argument, str) ends with the given target string (second argument, target).
This challenge can be solved with the .endsWith() method, which was introduced in ES2015. But for the purpose of this challenge, we would like you to use one of the JavaScript substring methods instead.

6.7 Repeat a String Repeat a String

Repeat a given string str (first argument) for num times (second argument). Return an empty string if num is not a positive number.

6.8 Truncate a String

Truncate a string (first argument) if it is longer than the given maximum string length (second argument). Return the truncated string with a ... ending.

6.9 Finders Keepers

Create a function that looks through an array ‘arr‘ and returns the first element in it that passes a ’truth test’. This means that given an element ‘x‘, the ’truth test’ is passed if ‘func(x)‘ is ‘true‘. If no element passes the test, return ‘undefined‘.

6.10 Boo who

Check if a value is classified as a boolean primitive. Return true or false.
Boolean primitives are true and false.

6.11 Title Case a Sentence

Return the provided string with the first letter of each word capitalized. Make sure the rest of the word is in lower case.
For the purpose of this exercise, you should also capitalize connecting words like "the" and "of".

6.12 Slice and Splice

You are given two arrays and an index.
Copy each element of the first array into the second array, in order.
Begin inserting elements at index n of the second array.
Return the resulting array. The input arrays should remain the same after the function runs.

6.13 Falsy Bouncer

Remove all falsy values from an array.
Falsy values in JavaScript are false, null, 0, "", undefined, and NaN.
Hint: Try converting each value to a Boolean.

6.14 Where do I Belong

Return the lowest index at which a value (second argument) should be inserted into an array (first argument) once it has been sorted. The returned value should be a number.
For example, getIndexToIns([1,2,3,4], 1.5) should return 1 because it is greater than 1 (index 0), but less than 2 (index 1).
Likewise, getIndexToIns([20,3,5], 19) should return 2 because once the array has been sorted it will look like [3,5,20] and 19 is less than 20 (index 2) and greater than 5 (index 1).

6.15 Mutations

Return true if the string in the first element of the array contains all of the letters of the string in the second element of the array.
For example, ["hello", "Hello"], should return true because all of the letters in the second string are present

in the first, ignoring case.
The arguments ["hello", "hey"] should return false because the string "hello" does not contain a "y".
Lastly, ["Alien", "line"], should return true because all of the letters in "line" are present in "Alien".

6.16 Chunky Monkey

Write a function that splits an array (first argument) into groups the length of size (second argument) and returns them as a two-dimensional array.

7 Object Oriented Programming

7.1 Create a Basic JavaScript Object

Think about things people see every day, like cars, shops, and birds. These are all objects: tangible things people can observe and interact with.
What are some qualities of these objects? A car has wheels. Shops sell items. Birds have wings.
These qualities, or properties, define what makes up an object. Note that similar objects share the same properties, but may have different values for those properties. For example, all cars have wheels, but not all cars have the same number of wheels.
Objects in JavaScript are used to model real-world objects, giving them properties and behavior just like their real-world counterparts. Here's an example using these concepts to create a duck object:
“‘js
let duck = {
name: "Aflac",
numLegs: 2
};
“‘
This duck object has two property/value pairs: a name of "Aflac" and a numLegs of 2.

7.2 Use Dot Notation to Access the Properties of an Object

The last challenge created an object with various properties. Now you'll see how to access the values of those properties. Here's an example:
“‘js
let duck = {
name: "Aflac",
numLegs: 2
};
console.log(duck.name);
// This prints "Aflac" to the console
“‘
Dot notation is used on the object name, duck, followed by the name of the property, name, to access the value of "Aflac".

7.3 Create a Method on an Object

Objects can have a special type of property, called a method.
Methods are properties that are functions. This adds different behavior to an object. Here is the duck example with a method:
“‘js
let duck = {
name: "Aflac",
numLegs: 2,
sayName: function() {return "The name of this duck is " + duck.name + ".";}
};
duck.sayName();
// Returns "The name of this duck is Aflac."
“‘
The example adds the sayName method, which is a function that returns a sentence giving the name of the duck.
Notice that the method accessed the name property in the return statement using duck.name. The next

challenge will cover another way to do this.

7.4 Make Code More Reusable with the this Keyword

The last challenge introduced a method to the duck object. It used duck.name dot notation to access the value for the name property within the return statement:
sayName: function() {return "The name of this duck is " + duck.name + ".";}
While this is a valid way to access the object's property, there is a pitfall here. If the variable name changes, any code referencing the original name would need to be updated as well. In a short object definition, it isn't a problem, but if an object has many references to its properties there is a greater chance for error.
A way to avoid these issues is with the this keyword:
“‘js
let duck = {
name: "Aflac",
numLegs: 2,
sayName: function() {return "The name of this duck is " + this.name + ".";}
};
“‘
this is a deep topic, and the above example is only one way to use it. In the current context, this refers to the object that the method is associated with: duck.
If the object's name is changed to mallard, it is not necessary to find all the references to duck in the code. It makes the code reusable and easier to read.

7.5 Define a Constructor Function

Constructors are functions that create new objects. They define properties and behaviors that will belong to the new object. Think of them as a blueprint for the creation of new objects.
Here is an example of a constructor:
“‘js
function Bird() {
this.name = "Albert";
this.color = "blue";
this.numLegs = 2;
}
“‘
This constructor defines a Bird object with properties name, color, and numLegs set to Albert, blue, and 2, respectively.
Constructors follow a few conventions:
Constructors are defined with a capitalized name to distinguish them from other functions that are not constructors.Constructors use the keyword this to set properties of the object they will create. Inside the constructor, this refers to the new object it will create.Constructors define properties and behaviors instead of returning a value as other functions might.

7.6 Use a Constructor to Create Objects

Here's the Bird constructor from the previous challenge:
“‘js
function Bird() {
this.name = "Albert";
this.color = "blue";
this.numLegs = 2;
// "this" inside the constructor always refers to the object being created

}
let blueBird = new Bird();
‘‘‘
Notice that the new operator is used when calling a constructor. This tells JavaScript to create a new instance of Bird called blueBird. Without the new operator, this inside the constructor would not point to the newly created object, giving unexpected results.
Now blueBird has all the properties defined inside the Bird constructor:
“‘js
blueBird.name; // => Albert
blueBird.color; // => blue
blueBird.numLegs; // => 2
‘‘‘
Just like any other object, its properties can be accessed and modified:
“‘js
blueBird.name = ’Elvira’;
blueBird.name; // => Elvira
‘‘‘

7.7 Extend Constructors to Receive Arguments

The Bird and Dog constructors from last challenge worked well. However, notice that all Birds that are created with the Bird constructor are automatically named Albert, are blue in color, and have two legs. What if you want birds with different values for name and color? It’s possible to change the properties of each bird manually but that would be a lot of work:
“‘js
let swan = new Bird();
swan.name = "Carlos";
swan.color = "white";
‘‘‘
Suppose you were writing a program to keep track of hundreds or even thousands of different birds in an aviary. It would take a lot of time to create all the birds, then change the properties to different values for every one.
To more easily create different Bird objects, you can design your Bird constructor to accept parameters:
“‘js
function Bird(name, color) {
this.name = name;
this.color = color;
this.numLegs = 2;
}
‘‘‘
Then pass in the values as arguments to define each unique bird into the Bird constructor:
let cardinal = new Bird("Bruce", "red");
This gives a new instance of Bird with name and color properties set to Bruce and red, respectively. The numLegs property is still set to 2.
The cardinal has these properties:
“‘js
cardinal.name // => Bruce
cardinal.color // => red
cardinal.numLegs // => 2
‘‘‘
The constructor is more flexible. It’s now possible to define the properties for each Bird at the time it is created, which is one way that JavaScript constructors are so useful. They group objects together based on shared characteristics and behavior and define a blueprint that automates their creation.

7.8 Verify an Object's Constructor with instanceof

Anytime a constructor function creates a new object, that object is said to be an instance of its constructor. JavaScript gives a convenient way to verify this with the instanceof operator. instanceof allows you to compare an object to a constructor, returning true or false based on whether or not that object was created with the constructor. Here's an example:
“‘js
let Bird = function(name, color) {
this.name = name;
this.color = color;
this.numLegs = 2;
}
let crow = new Bird("Alexis", "black");
crow instanceof Bird; // => true
“‘
If an object is created without using a constructor, instanceof will verify that it is not an instance of that constructor:
“‘js
let canary = {
name: "Mildred",
color: "Yellow",
numLegs: 2
};
canary instanceof Bird; // => false
“‘

7.9 Understand Own Properties

In the following example, the Bird constructor defines two properties: name and numLegs:
“‘js
function Bird(name) {
this.name = name;
this.numLegs = 2;
}
let duck = new Bird("Donald");
let canary = new Bird("Tweety");
“‘
name and numLegs are called own properties, because they are defined directly on the instance object. That means that duck and canary each has its own separate copy of these properties.
In fact every instance of Bird will have its own copy of these properties.
The following code adds all of the own properties of duck to the array ownProps:
“‘js
let ownProps = [];
for (let property in duck) {
if(duck.hasOwnProperty(property)) {
ownProps.push(property);
}
}
console.log(ownProps); // prints ["name", "numLegs"]
“‘

7.10 Use Prototype Properties to Reduce Duplicate Code

Since numLegs will probably have the same value for all instances of Bird, you essentially have a duplicated variable numLegs inside each Bird instance.
This may not be an issue when there are only two instances, but imagine if there are millions of instances. That would be a lot of duplicated variables.
A better way is to use Bird's prototype. Properties in the prototype are shared among ALL instances of Bird. Here's how to add numLegs to the Bird prototype:
“‘js
Bird.prototype.numLegs = 2;
“‘
Now all instances of Bird have the numLegs property.
“‘js
console.log(duck.numLegs); // prints 2
console.log(canary.numLegs); // prints 2
“‘
Since all instances automatically have the properties on the prototype, think of a prototype as a "recipe" for creating objects.
Note that the prototype for duck and canary is part of the Bird constructor as Bird.prototype. Nearly every object in JavaScript has a prototype property which is part of the constructor function that created it.

7.11 Iterate Over All Properties

You have now seen two kinds of properties: own properties and prototype properties. Own properties are defined directly on the object instance itself. And prototype properties are defined on the prototype.
“‘js
function Bird(name) {
this.name = name; //own property
}
Bird.prototype.numLegs = 2; // prototype property
let duck = new Bird("Donald");
“‘
Here is how you add duck's own properties to the array ownProps and prototype properties to the array prototypeProps:
“‘js
let ownProps = [];
let prototypeProps = [];
for (let property in duck) {
if(duck.hasOwnProperty(property)) {
ownProps.push(property);
} else {
prototypeProps.push(property);
}
}
console.log(ownProps); // prints ["name"]
console.log(prototypeProps); // prints ["numLegs"]
“‘

7.12 Understand the Constructor Property

There is a special constructor property located on the object instances duck and beagle that were created in the previous challenges:
“‘js

let duck = new Bird();
let beagle = new Dog();
console.log(duck.constructor === Bird); //prints true
console.log(beagle.constructor === Dog); //prints true
‘‘‘
Note that the constructor property is a reference to the constructor function that created the instance.
The advantage of the constructor property is that it’s possible to check for this property to find out what kind of object it is. Here’s an example of how this could be used:
“‘js
function joinBirdFraternity(candidate) {
if (candidate.constructor === Bird) {
return true;
} else {
return false;
}
}
‘‘‘
NoteSince the constructor property can be overwritten (which will be covered in the next two challenges) it’s generally better to use the instanceof method to check the type of an object.

7.13 Change the Prototype to a New Object

Up until now you have been adding properties to the prototype individually:
“‘js
Bird.prototype.numLegs = 2;
‘‘‘
This becomes tedious after more than a few properties.
“‘js
Bird.prototype.eat = function() {
console.log("nom nom nom");
}
Bird.prototype.describe = function() {
console.log("My name is " + this.name);
}
‘‘‘
A more efficient way is to set the prototype to a new object that already contains the properties. This way, the properties are added all at once:
“‘js
Bird.prototype = {
numLegs: 2,
eat: function() {
console.log("nom nom nom");
},
describe: function() {
console.log("My name is " + this.name);
}
};
‘‘‘

7.14 Remember to Set the Constructor Property when Changing the Prototype

There is one crucial side effect of manually setting the prototype to a new object. It erases the constructor property! This property can be used to check which constructor function created the instance, but since the property has been overwritten, it now gives false results:
“‘js
duck.constructor === Bird; // false -- Oops
duck.constructor === Object; // true, all objects inherit from Object.prototype
duck instanceof Bird; // true, still works
“‘
To fix this, whenever a prototype is manually set to a new object, remember to define the constructor property:
“‘js
Bird.prototype = {
constructor: Bird, // define the constructor property
numLegs: 2,
eat: function() {
console.log("nom nom nom");
},
describe: function() {
console.log("My name is " + this.name);
}
};
“‘

7.15 Understand Where an Object’s Prototype Comes From

Just like people inherit genes from their parents, an object inherits its prototype directly from the constructor function that created it. For example, here the Bird constructor creates the duck object:
“‘js
function Bird(name) {
this.name = name;
}
let duck = new Bird("Donald");
“‘
duck inherits its prototype from the Bird constructor function. You can show this relationship with the isPrototypeOf method:
“‘js
Bird.prototype.isPrototypeOf(duck);
// returns true
“‘

7.16 Understand the Prototype Chain

All objects in JavaScript (with a few exceptions) have a prototype. Also, an object’s prototype itself is an object.
“‘js
function Bird(name) {
this.name = name;
}
typeof Bird.prototype; // yields ’object’
“‘
Because a prototype is an object, a prototype can have its own prototype! In this case, the prototype of

Bird.prototype is Object.prototype:
“‘js
Object.prototype.isPrototypeOf(Bird.prototype); // returns true
“‘
How is this useful? You may recall the hasOwnProperty method from a previous challenge:
“‘js
let duck = new Bird("Donald");
duck.hasOwnProperty("name"); // yields true
“‘
The hasOwnProperty method is defined in Object.prototype, which can be accessed by Bird.prototype, which can then be accessed by duck. This is an example of the prototype chain.
In this prototype chain, Bird is the supertype for duck, while duck is the subtype. Object is a supertype for both Bird and duck.
Object is a supertype for all objects in JavaScript. Therefore, any object can use the hasOwnProperty method.

7.17 Use Inheritance So You Don’t Repeat Yourself

There’s a principle in programming called Don’t Repeat Yourself (DRY). The reason repeated code is a problem is because any change requires fixing code in multiple places. This usually means more work for programmers and more room for errors.
Notice in the example below that the describe method is shared by Bird and Dog:
“‘js
Bird.prototype = {
constructor: Bird,
describe: function() {
console.log("My name is " + this.name);
}
};
Dog.prototype = {
constructor: Dog,
describe: function() {
console.log("My name is " + this.name);
}
};
“‘
The describe method is repeated in two places. The code can be edited to follow the DRY principle by creating a supertype (or parent) called Animal:
“‘js
function Animal() { };
Animal.prototype = {
constructor: Animal,
describe: function() {
console.log("My name is " + this.name);
}
};
“‘
Since Animal includes the describe method, you can remove it from Bird and Dog:
“‘js
Bird.prototype = {
constructor: Bird
};
Dog.prototype = {
constructor: Dog

};
‘‘‘

7.18 Inherit Behaviors from a Supertype

In the previous challenge, you created a supertype called Animal that defined behaviors shared by all animals:
‘‘‘js
function Animal() { }
Animal.prototype.eat = function() {
console.log("nom nom nom");
};
‘‘‘
This and the next challenge will cover how to reuse Animal’s methods inside Bird and Dog without defining them again. It uses a technique called inheritance.
This challenge covers the first step: make an instance of the supertype (or parent).
You already know one way to create an instance of Animal using the new operator:
‘‘‘js
let animal = new Animal();
‘‘‘
There are some disadvantages when using this syntax for inheritance, which are too complex for the scope of this challenge. Instead, here’s an alternative approach without those disadvantages:
‘‘‘js
let animal = Object.create(Animal.prototype);
‘‘‘
Object.create(obj) creates a new object, and sets obj as the new object’s prototype. Recall that the prototype is like the "recipe" for creating an object. By setting the prototype of animal to be Animal’s prototype, you are effectively giving the animal instance the same "recipe" as any other instance of Animal.
‘‘‘js
animal.eat(); // prints "nom nom nom"
animal instanceof Animal; // => true
‘‘‘

7.19 Set the Child’s Prototype to an Instance of the Parent

In the previous challenge you saw the first step for inheriting behavior from the supertype (or parent) Animal: making a new instance of Animal.
This challenge covers the next step: set the prototype of the subtype (or child)—in this case, Bird—to be an instance of Animal.
‘‘‘js
Bird.prototype = Object.create(Animal.prototype);
‘‘‘
Remember that the prototype is like the "recipe" for creating an object. In a way, the recipe for Bird now includes all the key "ingredients" from Animal.
‘‘‘js
let duck = new Bird("Donald");
duck.eat(); // prints "nom nom nom"
‘‘‘
duck inherits all of Animal’s properties, including the eat method.

7.20 Reset an Inherited Constructor Property

When an object inherits its prototype from another object, it also inherits the supertype's constructor property.
Here's an example:
“‘js
function Bird() { }
Bird.prototype = Object.create(Animal.prototype);
let duck = new Bird();
duck.constructor // function Animal(){...}
“‘
But duck and all instances of Bird should show that they were constructed by Bird and not Animal. To do so, you can manually set Bird's constructor property to the Bird object:
“‘js
Bird.prototype.constructor = Bird;
duck.constructor // function Bird(){...}
“‘

7.21 Add Methods After Inheritance

A constructor function that inherits its prototype object from a supertype constructor function can still have its own methods in addition to inherited methods.
For example, Bird is a constructor that inherits its prototype from Animal:
“‘js
function Animal() { }
Animal.prototype.eat = function() {
console.log("nom nom nom");
};
function Bird() { }
Bird.prototype = Object.create(Animal.prototype);
Bird.prototype.constructor = Bird;
“‘
In addition to what is inherited from Animal, you want to add behavior that is unique to Bird objects. Here, Bird will get a fly() function. Functions are added to Bird's prototype the same way as any constructor function:
“‘js
Bird.prototype.fly = function() {
console.log("I'm flying!");
};
“‘
Now instances of Bird will have both eat() and fly() methods:
“‘js
let duck = new Bird();
duck.eat(); // prints "nom nom nom"
duck.fly(); // prints "I'm flying!"
“‘

7.22 Override Inherited Methods

In previous lessons, you learned that an object can inherit its behavior (methods) from another object by referencing its prototype object:
“‘js
ChildObject.prototype = Object.create(ParentObject.prototype);

```
Then the ChildObject received its own methods by chaining them onto its prototype:
```js
ChildObject.prototype.methodName = function() {...};
```
It's possible to override an inherited method. It's done the same way - by adding a method to ChildObject.prototype using the same method name as the one to override.
Here's an example of Bird overriding the eat() method inherited from Animal:
```js
function Animal() { }
Animal.prototype.eat = function() {
return "nom nom nom";
};
function Bird() { }
// Inherit all methods from Animal
Bird.prototype = Object.create(Animal.prototype);
// Bird.eat() overrides Animal.eat()
Bird.prototype.eat = function() {
return "peck peck peck";
};
```
If you have an instance let duck = new Bird(); and you call duck.eat(), this is how JavaScript looks for the method on duck's prototype chain:
1. duck => Is eat() defined here? No.
2. Bird => Is eat() defined here? => Yes. Execute it and stop searching.
3. Animal => eat() is also defined, but JavaScript stopped searching before reaching this level.
4. Object => JavaScript stopped searching before reaching this level.

7.23 Use a Mixin to Add Common Behavior Between Unrelated Objects

As you have seen, behavior is shared through inheritance. However, there are cases when inheritance is not the best solution. Inheritance does not work well for unrelated objects like Bird and Airplane. They can both fly, but a Bird is not a type of Airplane and vice versa.
For unrelated objects, it's better to use mixins. A mixin allows other objects to use a collection of functions.
```js
let flyMixin = function(obj) {
obj.fly = function() {
console.log("Flying, wooosh!");
}
};
```
The flyMixin takes any object and gives it the fly method.
```js
let bird = {
name: "Donald",
numLegs: 2
};
let plane = {
model: "777",
numPassengers: 524
};
flyMixin(bird);
flyMixin(plane);
```
```

```
Here bird and plane are passed into flyMixin, which then assigns the fly function to each object. Now bird and plane can both fly:
```js
bird.fly(); // prints "Flying, wooosh!"
plane.fly(); // prints "Flying, wooosh!"
```
Note how the mixin allows for the same fly method to be reused by unrelated objects bird and plane.

7.24 Use Closure to Protect Properties Within an Object from Being Modified Externally

In the previous challenge, bird had a public property name. It is considered public because it can be accessed and changed outside of bird's definition.
```js
bird.name = "Duffy";
```
Therefore, any part of your code can easily change the name of bird to any value. Think about things like passwords and bank accounts being easily changeable by any part of your codebase. That could cause a lot of issues.
The simplest way to make this public property private is by creating a variable within the constructor function. This changes the scope of that variable to be within the constructor function versus available globally. This way, the variable can only be accessed and changed by methods also within the constructor function.
```js
function Bird() {
let hatchedEgg = 10; // private variable
/* publicly available method that a bird object can use */
this.getHatchedEggCount = function() {
return hatchedEgg;
};
}
let ducky = new Bird();
ducky.getHatchedEggCount(); // returns 10
```
Here getHatchedEggCount is a privileged method, because it has access to the private variable hatchedEgg. This is possible because hatchedEgg is declared in the same context as getHatchedEggCount. In JavaScript, a function always has access to the context in which it was created. This is called closure.

7.25 Understand the Immediately Invoked Function Expression (IIFE)

A common pattern in JavaScript is to execute a function as soon as it is declared:
```js
(function () {
console.log("Chirp, chirp!");
})(); // this is an anonymous function expression that executes right away
// Outputs "Chirp, chirp!" immediately
```
Note that the function has no name and is not stored in a variable. The two parentheses () at the end of the function expression cause it to be immediately executed or invoked. This pattern is known as an immediately invoked function expression or IIFE.
```

7.26 Use an IIFE to Create a Module

An immediately invoked function expression (IIFE) is often used to group related functionality into a single object or module. For example, an earlier challenge defined two mixins:
“‘js
function glideMixin(obj) {
obj.glide = function() {
console.log("Gliding on the water");
};
}
function flyMixin(obj) {
obj.fly = function() {
console.log("Flying, wooosh!");
};
}
“‘
We can group these mixins into a module as follows:
“‘js
let motionModule = (function () {
return {
glideMixin: function(obj) {
obj.glide = function() {
console.log("Gliding on the water");
};
},
flyMixin: function(obj) {
obj.fly = function() {
console.log("Flying, wooosh!");
};
}
}
})(); // The two parentheses cause the function to be immediately invoked
“‘
Note that you have an immediately invoked function expression (IIFE) that returns an object motionModule. This returned object contains all of the mixin behaviors as properties of the object.
The advantage of the module pattern is that all of the motion behaviors can be packaged into a single object that can then be used by other parts of your code. Here is an example using it:
“‘js
motionModule.glideMixin(duck);
duck.glide();
“‘

8 Functional Programming

8.1 Learn About Functional Programming

Functional programming is a style of programming where solutions are simple, isolated functions, without any side effects outside of the function scope.
INPUT -> PROCESS -> OUTPUT
Functional programming is about:
1) Isolated functions - there is no dependence on the state of the program, which includes global variables that are subject to change
2) Pure functions - the same input always gives the same output
3) Functions with limited side effects - any changes, or mutations, to the state of the program outside the function are carefully controlled

8.2 Understand Functional Programming Terminology

The FCC Team had a mood swing and now wants two types of tea: green tea and black tea. General Fact: Client mood swings are pretty common.
With that information, we'll need to revisit the getTea function from last challenge to handle various tea requests. We can modify getTea to accept a function as a parameter to be able to change the type of tea it prepares. This makes getTea more flexible, and gives the programmer more control when client requests change.
But first, let's cover some functional terminology:
Callbacks are the functions that are slipped or passed into another function to decide the invocation of that function. You may have seen them passed to other methods, for example in filter, the callback function tells JavaScript the criteria for how to filter an array.
Functions that can be assigned to a variable, passed into another function, or returned from another function just like any other normal value, are called first class functions. In JavaScript, all functions are first class functions.
The functions that take a function as an argument, or return a function as a return value are called higher order functions.
When the functions are passed in to another function or returned from another function, then those functions which gets passed in or returned can be called a lambda.

8.3 Understand the Hazards of Using Imperative Code

Functional programming is a good habit. It keeps your code easy to manage, and saves you from sneaky bugs. But before we get there, let's look at an imperative approach to programming to highlight where you may have issues.
In English (and many other languages), the imperative tense is used to give commands. Similarly, an imperative style in programming is one that gives the computer a set of statements to perform a task.
Often the statements change the state of the program, like updating global variables. A classic example is writing a for loop that gives exact directions to iterate over the indices of an array.
In contrast, functional programming is a form of declarative programming. You tell the computer what you want done by calling a method or function.
JavaScript offers many predefined methods that handle common tasks so you don't need to write out how the computer should perform them. For example, instead of using the for loop mentioned above, you could call the map method which handles the details of iterating over an array. This helps to avoid semantic errors, like the "Off By One Errors" that were covered in the Debugging section.
Consider the scenario: you are browsing the web in your browser, and want to track the tabs you have opened. Let's try to model this using some simple object-oriented code.
A Window object is made up of tabs, and you usually have more than one Window open. The titles of each open site in each Window object is held in an array. After working in the browser (opening new tabs, merging

windows, and closing tabs), you want to print the tabs that are still open. Closed tabs are removed from the array and new tabs (for simplicity) get added to the end of it.
The code editor shows an implementation of this functionality with functions for tabOpen(), tabClose(), and join(). The array tabs is part of the Window object that stores the name of the open pages.

8.4 Avoid Mutations and Side Effects Using Functional Programming

If you haven't already figured it out, the issue in the previous challenge was with the splice call in the tabClose() function. Unfortunately, splice changes the original array it is called on, so the second call to it used a modified array, and gave unexpected results.
This is a small example of a much larger pattern - you call a function on a variable, array, or an object, and the function changes the variable or something in the object.
One of the core principles of functional programming is to not change things. Changes lead to bugs. It's easier to prevent bugs knowing that your functions don't change anything, including the function arguments or any global variable.
The previous example didn't have any complicated operations but the splice method changed the original array, and resulted in a bug.
Recall that in functional programming, changing or altering things is called mutation, and the outcome is called a side effect. A function, ideally, should be a pure function, meaning that it does not cause any side effects.
Let's try to master this discipline and not alter any variable or object in our code.

8.5 Pass Arguments to Avoid External Dependence in a Function

The last challenge was a step closer to functional programming principles, but there is still something missing. We didn't alter the global variable value, but the function incrementer would not work without the global variable fixedValue being there.
Another principle of functional programming is to always declare your dependencies explicitly. This means if a function depends on a variable or object being present, then pass that variable or object directly into the function as an argument.
There are several good consequences from this principle. The function is easier to test, you know exactly what input it takes, and it won't depend on anything else in your program.
This can give you more confidence when you alter, remove, or add new code. You would know what you can or cannot change and you can see where the potential traps are.
Finally, the function would always produce the same output for the same set of inputs, no matter what part of the code executes it.

8.6 Refactor Global Variables Out of Functions

So far, we have seen two distinct principles for functional programming:
1) Don't alter a variable or object - create new variables and objects and return them if need be from a function.
2) Declare function arguments - any computation inside a function depends only on the arguments, and not on any global object or variable.
Adding one to a number is not very exciting, but we can apply these principles when working with arrays or more complex objects.

8.7 Use the map Method to Extract Data from an Array

So far we have learned to use pure functions to avoid side effects in a program. Also, we have seen the value in having a function only depend on its input arguments.

This is only the beginning. As its name suggests, functional programming is centered around a theory of functions.
It would make sense to be able to pass them as arguments to other functions, and return a function from another function. Functions are considered first class objects in JavaScript, which means they can be used like any other object. They can be saved in variables, stored in an object, or passed as function arguments.
Let's start with some simple array functions, which are methods on the array object prototype. In this exercise we are looking at Array.prototype.map(), or more simply map.
The map method iterates over each item in an array and returns a new array containing the results of calling the callback function on each element. It does this without mutating the original array.
When the callback is used, it is passed three arguments. The first argument is the current element being processed. The second is the index of that element and the third is the array upon which the map method was called.
See below for an example using the map method on the users array to return a new array containing only the names of the users as elements. For simplicity, the example only uses the first argument of the callback.
“‘js
const users = [
{ name: 'John', age: 34 },
{ name: 'Amy', age: 20 },
{ name: 'camperCat', age: 10 }
];
const names = users.map(user => user.name);
console.log(names); // ['John', 'Amy', 'camperCat']
“‘

8.8 Implement map on a Prototype

As you have seen from applying Array.prototype.map(), or simply map() earlier, the map method returns an array of the same length as the one it was called on. It also doesn't alter the original array, as long as its callback function doesn't.
In other words, map is a pure function, and its output depends solely on its inputs. Plus, it takes another function as its argument.
You might learn a lot about the map method if you implement your own version of it. It is recommended you use a for loop or Array.prototype.forEach().

8.9 Use the filter Method to Extract Data from an Array

Another useful array function is Array.prototype.filter(), or simply filter().
filter calls a function on each element of an array and returns a new array containing only the elements for which that function returns true. In other words, it filters the array, based on the function passed to it. Like map, it does this without needing to modify the original array.
The callback function accepts three arguments. The first argument is the current element being processed. The second is the index of that element and the third is the array upon which the filter method was called.
See below for an example using the filter method on the users array to return a new array containing only the users under the age of 30. For simplicity, the example only uses the first argument of the callback.
“‘js
const users = [
{ name: 'John', age: 34 },
{ name: 'Amy', age: 20 },
{ name: 'camperCat', age: 10 }
];
const usersUnder30 = users.filter(user => user.age < 30);
console.log(usersUnder30); // [{ name: 'Amy', age: 20 }, { name: 'camperCat', age: 10 }]

```

## 8.10 Implement the filter Method on a Prototype

You might learn a lot about the filter method if you implement your own version of it. It is recommended you use a for loop or Array.prototype.forEach().

## 8.11 Return Part of an Array Using the slice Method

The slice method returns a copy of certain elements of an array. It can take two arguments, the first gives the index of where to begin the slice, the second is the index for where to end the slice (and it's non-inclusive). If the arguments are not provided, the default is to start at the beginning of the array through the end, which is an easy way to make a copy of the entire array. The slice method does not mutate the original array, but returns a new one.
Here's an example:
```js
var arr = ["Cat", "Dog", "Tiger", "Zebra"];
var newArray = arr.slice(1, 3);
// Sets newArray to ["Dog", "Tiger"]
```

8.12 Remove Elements from an Array Using slice Instead of splice

A common pattern while working with arrays is when you want to remove items and keep the rest of the array. JavaScript offers the splice method for this, which takes arguments for the index of where to start removing items, then the number of items to remove. If the second argument is not provided, the default is to remove items through the end. However, the splice method mutates the original array it is called on.
Here's an example:
```js
var cities = ["Chicago", "Delhi", "Islamabad", "London", "Berlin"];
cities.splice(3, 1); // Returns "London" and deletes it from the cities array
// cities is now ["Chicago", "Delhi", "Islamabad", "Berlin"]
```
As we saw in the last challenge, the slice method does not mutate the original array, but returns a new one which can be saved into a variable. Recall that the slice method takes two arguments for the indices to begin and end the slice (the end is non-inclusive), and returns those items in a new array. Using the slice method instead of splice helps to avoid any array-mutating side effects.

8.13 Combine Two Arrays Using the concat Method

Concatenation means to join items end to end. JavaScript offers the concat method for both strings and arrays that work in the same way. For arrays, the method is called on one, then another array is provided as the argument to concat, which is added to the end of the first array. It returns a new array and does not mutate either of the original arrays. Here's an example:
```js
[1, 2, 3].concat([4, 5, 6]);
// Returns a new array [1, 2, 3, 4, 5, 6]
```
```

8.14 Add Elements to the End of an Array Using concat Instead of push

Functional programming is all about creating and using non-mutating functions.
The last challenge introduced the concat method as a way to combine arrays into a new one without mutating the original arrays. Compare concat to the push method. Push adds an item to the end of the same array it is called on, which mutates that array. Here's an example:
“‘js
var arr = [1, 2, 3];
arr.push([4, 5, 6]);
// arr is changed to [1, 2, 3, [4, 5, 6]]
// Not the functional programming way
“‘
Concat offers a way to add new items to the end of an array without any mutating side effects.

8.15 Use the reduce Method to Analyze Data

Array.prototype.reduce(), or simply reduce(), is the most general of all array operations in JavaScript. You can solve almost any array processing problem using the reduce method.
The reduce method allows for more general forms of array processing, and it's possible to show that both filter and map can be derived as special applications of reduce.
The reduce method iterates over each item in an array and returns a single value (i.e. string, number, object, array). This is achieved via a callback function that is called on each iteration.
The callback function accepts four arguments. The first argument is known as the accumulator, which gets assigned the return value of the callback function from the previous iteration, the second is the current element being processed, the third is the index of that element and the fourth is the array upon which reduce is called.
In addition to the callback function, reduce has an additional parameter which takes an initial value for the accumulator. If this second parameter is not used, then the first iteration is skipped and the second iteration gets passed the first element of the array as the accumulator.
See below for an example using reduce on the users array to return the sum of all the users' ages. For simplicity, the example only uses the first and second arguments.
“‘js
const users = [
{ name: 'John', age: 34 },
{ name: 'Amy', age: 20 },
{ name: 'camperCat', age: 10 }
];
const sumOfAges = users.reduce((sum, user) => sum + user.age, 0);
console.log(sumOfAges); // 64
“‘
In another example, see how an object can be returned containing the names of the users as properties with their ages as values.
“‘js
const users = [
{ name: 'John', age: 34 },
{ name: 'Amy', age: 20 },
{ name: 'camperCat', age: 10 }
];
const usersObj = users.reduce((obj, user) => {
obj[user.name] = user.age;
return obj;
}, {});
console.log(usersObj); // { John: 34, Amy: 20, camperCat: 10 }
“‘

8.16 Use Higher-Order Functions map, filter, or reduce to Solve a Complex Problem

Now that you have worked through a few challenges using higher-order functions like map(), filter(), and reduce(), you now get to apply them to solve a more complex challenge.

8.17 Sort an Array Alphabetically using the sort Method

The sort method sorts the elements of an array according to the callback function.
For example:
“‘js
function ascendingOrder(arr) {
return arr.sort(function(a, b) {
return a - b;
});
}
ascendingOrder([1, 5, 2, 3, 4]);
// Returns [1, 2, 3, 4, 5]
function reverseAlpha(arr) {
return arr.sort(function(a, b) {
return a === b ? 0 : a < b ? 1 : -1;
});
}
reverseAlpha(['l', 'h', 'z', 'b', 's']);
// Returns ['z', 's', 'l', 'h', 'b']
“‘
JavaScript's default sorting method is by string Unicode point value, which may return unexpected results. Therefore, it is encouraged to provide a callback function to specify how to sort the array items. When such a callback function, normally called compareFunction, is supplied, the array elements are sorted according to the return value of the compareFunction:
If compareFunction(a,b) returns a value less than 0 for two elements a and b, then a will come before b.
If compareFunction(a,b) returns a value greater than 0 for two elements a and b, then b will come before a.
If compareFunction(a,b) returns a value equal to 0 for two elements a and b, then a and b will remain unchanged.

8.18 Return a Sorted Array Without Changing the Original Array

A side effect of the sort method is that it changes the order of the elements in the original array. In other words, it mutates the array in place. One way to avoid this is to first concatenate an empty array to the one being sorted (remember that slice and concat return a new array), then run the sort method.

8.19 Split a String into an Array Using the split Method

The split method splits a string into an array of strings. It takes an argument for the delimiter, which can be a character to use to break up the string or a regular expression. For example, if the delimiter is a space, you get an array of words, and if the delimiter is an empty string, you get an array of each character in the string.
Here are two examples that split one string by spaces, then another by digits using a regular expression:
“‘js
var str = "Hello World";

```
var bySpace = str.split(" ");
// Sets bySpace to ["Hello", "World"]
var otherString = "How9are7you2today";
var byDigits = otherString.split(/\d/);
// Sets byDigits to ["How", "are", "you", "today"]
```

Since strings are immutable, the split method makes it easier to work with them.

8.20 Combine an Array into a String Using the join Method

The join method is used to join the elements of an array together to create a string. It takes an argument for the delimiter that is used to separate the array elements in the string.
Here's an example:

```js
var arr = ["Hello", "World"];
var str = arr.join(" ");
// Sets str to "Hello World"
```

8.21 Apply Functional Programming to Convert Strings to URL Slugs

The last several challenges covered a number of useful array and string methods that follow functional programming principles. We've also learned about reduce, which is a powerful method used to reduce problems to simpler forms. From computing averages to sorting, any array operation can be achieved by applying it. Recall that map and filter are special cases of reduce.
Let's combine what we've learned to solve a practical problem.
Many content management sites (CMS) have the titles of a post added to part of the URL for simple bookmarking purposes. For example, if you write a Medium post titled "Stop Using Reduce", it's likely the URL would have some form of the title string in it (".../stop-using-reduce"). You may have already noticed this on the freeCodeCamp site.

8.22 Use the every Method to Check that Every Element in an Array Meets a Criteria

The every method works with arrays to check if every element passes a particular test. It returns a Boolean value - true if all values meet the criteria, false if not.
For example, the following code would check if every element in the numbers array is less than 10:

```js
var numbers = [1, 5, 8, 0, 10, 11];
numbers.every(function(currentValue) {
return currentValue < 10;
});
// Returns false
```

8.23 Use the some Method to Check that Any Elements in an Array Meet a Criteria

The some method works with arrays to check if any element passes a particular test. It returns a Boolean value - true if any of the values meet the criteria, false if not.

For example, the following code would check if any element in the numbers array is less than 10:
“‘js
var numbers = [10, 50, 8, 220, 110, 11];
numbers.some(function(currentValue) {
return currentValue < 10;
});
// Returns true
“‘

8.24 Introduction to Currying and Partial Application

The arity of a function is the number of arguments it requires. Currying a function means to convert a function of N arity into N functions of arity 1.
In other words, it restructures a function so it takes one argument, then returns another function that takes the next argument, and so on.
Here's an example:
“‘js
//Un-curried function
function unCurried(x, y) {
return x + y;
}
//Curried function
function curried(x) {
return function(y) {
return x + y;
}
}
//Alternative using ES6
const curried = x => y => x + y
curried(1)(2) // Returns 3
“‘
This is useful in your program if you can't supply all the arguments to a function at one time. You can save each function call into a variable, which will hold the returned function reference that takes the next argument when it's available. Here's an example using the curried function in the example above:
“‘js
// Call a curried function in parts:
var funcForY = curried(1);
console.log(funcForY(2)); // Prints 3
“‘
Similarly, partial application can be described as applying a few arguments to a function at a time and returning another function that is applied to more arguments.
Here's an example:
“‘js
//Impartial function
function impartial(x, y, z) {
return x + y + z;
}
var partialFn = impartial.bind(this, 1, 2);
partialFn(10); // Returns 13
“‘

9 Intermediate Algorithm Scripting

9.1 Sum All Numbers in a Range

We'll pass you an array of two numbers. Return the sum of those two numbers plus the sum of all the numbers between them. The lowest number will not always come first.
For example, sumAll([4,1]) should return 10 because sum of all the numbers between 1 and 4 (both inclusive) is 10.

9.2 Diff Two Arrays

Compare two arrays and return a new array with any items only found in one of the two given arrays, but not both. In other words, return the symmetric difference of the two arrays.
NoteYou can return the array with its elements in any order.

9.3 Seek and Destroy

You will be provided with an initial array (the first argument in the destroyer function), followed by one or more arguments. Remove all elements from the initial array that are of the same value as these arguments.
Note You have to use the arguments object.

9.4 Wherefore art thou

Make a function that looks through an array of objects (first argument) and returns an array of all objects that have matching name and value pairs (second argument). Each name and value pair of the source object has to be present in the object from the collection if it is to be included in the returned array.
For example, if the first argument is [{ first: "Romeo", last: "Montague" }, { first: "Mercutio", last: null }, { first: "Tybalt", last: "Capulet" }], and the second argument is { last: "Capulet" }, then you must return the third object from the array (the first argument), because it contains the name and its value, that was passed on as the second argument.

9.5 Spinal Tap Case

Convert a string to spinal case. Spinal case is all-lowercase-words-joined-by-dashes.

9.6 Pig Latin

Pig Latin is a way of altering English Words. The rules are as follows:
- If a word begins with a consonant, take the first consonant or consonant cluster, move it to the end of the word, and add "ay" to it.
- If a word begins with a vowel, just add "way" at the end.

9.7 Search and Replace

Perform a search and replace on the sentence using the arguments provided and return the new sentence.
First argument is the sentence to perform the search and replace on.
Second argument is the word that you will be replacing (before).
Third argument is what you will be replacing the second argument with (after).
Note Preserve the case of the first character in the original word when you are replacing it. For example if

you mean to replace the word "Book" with the word "dog", it should be replaced as "Dog"

9.8 DNA Pairing

The DNA strand is missing the pairing element. Take each character, get its pair, and return the results as a 2d array.
Base pairs are a pair of AT and CG. Match the missing element to the provided character.
Return the provided character as the first element in each array.
For example, for the input GCG, return [["G", "C"], ["C","G"],["G", "C"]]
The character and its pair are paired up in an array, and all the arrays are grouped into one encapsulating array.

9.9 Missing letters

Find the missing letter in the passed letter range and return it.
If all letters are present in the range, return undefined.

9.10 Sorted Union

Write a function that takes two or more arrays and returns a new array of unique values in the order of the original provided arrays.
In other words, all values present from all arrays should be included in their original order, but with no duplicates in the final array.
The unique numbers should be sorted by their original order, but the final array should not be sorted in numerical order.
Check the assertion tests for examples.

9.11 Convert HTML Entities

Convert the characters &, <, >, " (double quote), and ' (apostrophe), in a string to their corresponding HTML entities.

9.12 Sum All Odd Fibonacci Numbers

Given a positive integer num, return the sum of all odd Fibonacci numbers that are less than or equal to num.
The first two numbers in the Fibonacci sequence are 1 and 1. Every additional number in the sequence is the sum of the two previous numbers. The first six numbers of the Fibonacci sequence are 1, 1, 2, 3, 5 and 8.
For example, sumFibs(10) should return 10 because all odd Fibonacci numbers less than or equal to 10 are 1, 1, 3, and 5.

9.13 Sum All Primes

A prime number is a whole number greater than 1 with exactly two divisors: 1 and itself. For example, 2 is a prime number because it is only divisible by 1 and 2. In contrast, 4 is not prime since it is divisible by 1, 2 and 4.
Rewrite ‘sumPrimes‘ so it returns the sum of all prime numbers that are less than or equal to num.

9.14 Smallest Common Multiple

Find the smallest common multiple of the provided parameters that can be evenly divided by both, as well as by all sequential numbers in the range between these parameters.
The range will be an array of two numbers that will not necessarily be in numerical order.
For example, if given 1 and 3, find the smallest common multiple of both 1 and 3 that is also evenly divisible by all numbers between 1 and 3. The answer here would be 6.

9.15 Drop it

Given the array arr, iterate through and remove each element starting from the first element (the 0 index) until the function func returns true when the iterated element is passed through it.
Then return the rest of the array once the condition is satisfied, otherwise, arr should be returned as an empty array.

9.16 Steamroller

Flatten a nested array. You must account for varying levels of nesting.

9.17 Binary Agents

Return an English translated sentence of the passed binary string.
The binary string will be space separated.

9.18 Everything Be True

Check if the predicate (second argument) is truthy on all elements of a collection (first argument).
In other words, you are given an array collection of objects. The predicate pre will be an object property and you need to return true if its value is truthy. Otherwise, return false.
In JavaScript, truthy values are values that translate to true when evaluated in a Boolean context.
Remember, you can access object properties through either dot notation or [] notation.

9.19 Arguments Optional

Create a function that sums two arguments together. If only one argument is provided, then return a function that expects one argument and returns the sum.
For example, addTogether(2, 3) should return 5, and addTogether(2) should return a function.
Calling this returned function with a single argument will then return the sum:
var sumTwoAnd = addTogether(2);
sumTwoAnd(3) returns 5.
If either argument isn't a valid number, return undefined.

9.20 Make a Person

Fill in the object constructor with the following methods below:
“‘js
getFirstName()
getLastName()
getFullName()

setFirstName(first)
setLastName(last)
setFullName(firstAndLast)
‘‘‘

Run the tests to see the expected output for each method.
The methods that take an argument must accept only one argument and it has to be a string.
These methods must be the only available means of interacting with the object.

9.21 Map the Debris

Return a new array that transforms the elements' average altitude into their orbital periods (in seconds).
The array will contain objects in the format {name: 'name', avgAlt: avgAlt}.
You can read about orbital periods on Wikipedia.
The values should be rounded to the nearest whole number. The body being orbited is Earth.
The radius of the earth is 6367.4447 kilometers, and the GM value of earth is 398600.4418 km3s-2.

10 Javascript Algorithms And Data Structures Projects

10.1 Palindrome Checker

Return true if the given string is a palindrome. Otherwise, return false.
A palindrome is a word or sentence that's spelled the same way both forward and backward, ignoring punctuation, case, and spacing.
NoteYou'll need to remove all non-alphanumeric characters (punctuation, spaces and symbols) and turn everything into the same case (lower or upper case) in order to check for palindromes.
We'll pass strings with varying formats, such as "racecar", "RaceCar", and "race CAR" among others.
We'll also pass strings with special symbols, such as "2A3*3a2", "2A3 3a2", and "2_A3*3#A2".

10.2 Roman Numeral Converter

Convert the given number into a roman numeral.
All roman numerals answers should be provided in upper-case.

10.3 Caesars Cipher

One of the simplest and most widely known ciphers is a Caesar cipher, also known as a shift cipher. In a shift cipher the meanings of the letters are shifted by some set amount.
A common modern use is the ROT13 cipher, where the values of the letters are shifted by 13 places. Thus 'A' <-> 'N', 'B' <-> 'O' and so on.
Write a function which takes a ROT13 encoded string as input and returns a decoded string.
All letters will be uppercase. Do not transform any non-alphabetic character (i.e. spaces, punctuation), but do pass them on.

10.4 Telephone Number Validator

Return true if the passed string looks like a valid US phone number.
The user may fill out the form field any way they choose as long as it has the format of a valid US number. The following are examples of valid formats for US numbers (refer to the tests below for other variants):
555-555-5555(555)555-5555(555) 555-5555555 555 555555555555551 555 555 5555
For this challenge you will be presented with a string such as 800-692-7753 or 8oo-six427676;laskdjf. Your job is to validate or reject the US phone number based on any combination of the formats provided above. The area code is required. If the country code is provided, you must confirm that the country code is 1. Return true if the string is a valid US phone number; otherwise return false.

10.5 Cash Register

Design a cash register drawer function checkCashRegister() that accepts purchase price as the first argument (price), payment as the second argument (cash), and cash-in-drawer (cid) as the third argument.
cid is a 2D array listing available currency.
The checkCashRegister() function should always return an object with a status key and a change key.
Return {status: "INSUFFICIENT_FUNDS", change: []} if cash-in-drawer is less than the change due, or if you cannot return the exact change.
Return {status: "CLOSED", change: [...]} with cash-in-drawer as the value for the key change if it is equal to the change due.
Otherwise, return {status: "OPEN", change: [...]}, with the change due in coins and bills, sorted in highest to lowest order, as the value of the change key.

Currency UnitAmountPenny$0.01 (PENNY)Nickel$0.05 (NICKEL)Dime$0.1 (DIME)Quarter$0.25 (QUARTER)Dollar$1 (ONE)Five Dollars$5 (FIVE)Ten Dollars$10 (TEN)Twenty Dollars$20 (TWENTY)One-hundred Dollars$100 (ONE HUNDRED)
See below for an example of a cash-in-drawer array:
“‘js
[
["PENNY", 1.01],
["NICKEL", 2.05],
["DIME", 3.1],
["QUARTER", 4.25],
["ONE", 90],
["FIVE", 55],
["TEN", 20],
["TWENTY", 60],
["ONE HUNDRED", 100]
]
“‘