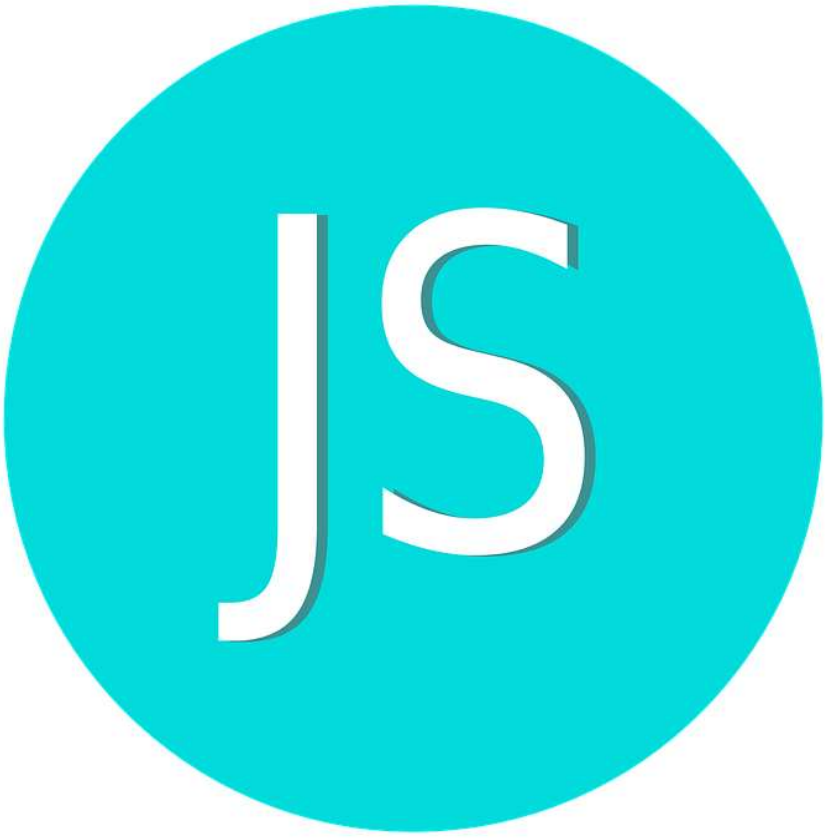
Student Guide to Coding

JavaScript



Javascript Algorithms And Data Structures Certification (300 hours)

Introduction to JavaScript

JavaScript is a high-level programming language that all modern web browsers support. It is also one of the core technologies of the web, along with HTML and CSS that you may have learned previously. This section will cover basic JavaScript programming concepts, which range from variables and arithmetic to objects and loops.

Commenting your Code

In JavaScript you can comment, your code like you can in HTML. There are two ways to create a comment in JavaScript. You have in-line comments and multi-line comments.

Ex. *// This is an example of an in-line comment*
Ex. */* This is a* *Multi-line comment */*

Declaring Variables

A Variable is a container that contains data. JavaScript provides seven different data types you can save within a variable. Undefined, null, boolean, string, symbol, number, and object. Variables are very similar to x and y variables in mathematics. They are just simple names to represent the data we want to refer to, but they differ in that they can store different values at different times.

To create a variable we use the keyword var and then the name of our variable.

```
var name;  // This variable is currently undefined
```

Storing Values with the assignment operator

In JavaScript you can store a value within a variable with the assignment operator. It is common to initialize a variable to an initial value in the same line as its declared.

```
myNum = 5; // This assigns the number 5 to myNum
```

myString = "I ran 5 miles" // Everything within quotes is considered a string
myBool = true // Boolean expressions are either true or false

Understanding Uninitialized Variables

If you do not give an initial value to a variable, it will be given a value of undefined. If you do mathematical operations on an undefined variable, your result will be NaN (Not a Number). If you concatenate a string with undefined variable, you will get a literal string of undefined.

Case Sensitivity in Variables

In JavaScript variables and functions are case sensitive, which means capitalization matters. MYVAR does not equal MyVar or myvar.

The best practice in naming variables in JavaScript is using camelCase. In camelCase, multi-word variable names have the first word in lowercase and the first letter of each subsequent word is capitalized.

Examples

```
var  someVariable;
var  anotherVariableName;
var thisVariableNameIsSoLong;
```

Basic Mathematics with JavaScript

When you have numeric data you can apply basic mathematics to that data, which includes addition, subtraction, multiplication, and division.

Addition	myAdd = 8 + 8;
Subtraction	mySub = 12 -6;
Multiplication	myMult = 13 * 13;
Division	myDiv = 16/2;

Incrementing and Decrementing a number with JavaScript

If you wish to easily increment a number by 1 you can use the ++ Operator. So instead of using i = i + 1;. You will use i++; instead. This also goes with decrementing, a number by 1. Instead of i = i - 1;. You will use i--;.

Incrementing by 1	i++;
Decrementing by 1	i--;

Creating Decimal Numbers

Decimal numbers can be stored in variables as well. They are known as floating point numbers or floats.

```
var myDec = 3.14;
```

Much like integers, you can apply the same mathematics to decimals as well.

Addition	myAdd = 2.3 + 3.5;
Subtraction	mySub = 6.3 - 2.5;
Multiplication	myMult = 5.5 * 2.6;
Division	myDiv = 4.4 / 2.0;

Finding a Remainder

The remainder operator % gives the remainder of the division of two numbers.

17 % 2 = 1
48 % 2 = 0

Compound Assignment with Augmented Mathematics

It is common to do some form of mathematics to an existing variable. So lines of code like this are common to see.

```
myVar = myVar + 5;
```

Because this is so common, operators to handle this type of mathematics have been created.

myVar = myVar + 5;	myVar += 5;
myVar = myVar - 5;	myVar -= 5;

myVar = myVar * 5;	myVar *= 5;
myVar = myVar / 5;	myVar /= 5;

Declaring String Variables

String Variables are enclosed within single or double quotes.

```
var firstName = “John”;
var lastName = ‘Doe’;
```

So a String needs to start and end with quotes, but how do we handle a situation when we need literal quotes within our string? In order to solve this problem there is something known as an escape.

We can add quotes to our string by adding / in front of the quotes we are adding to the string.

```
var myString = “I love \”Double Quotes \”.”;
```

Escape Sequences in Strings

There are other characters that need to be escaped to use within a string besides quotes. The following chart will show you those escapes and what each one does.

\’	Single Quote
\”	Double Quote
\\	backslash
\n	newline
\r	Carriage return
\t	tab
\b	backspace
\f	Form feed

Concatenation

Concatenation is combining two strings together. This is done by using the addition sign.

```
var myVar = “Hello,” + “ World”
```

This line of code will give us a variable myVar that holds the string “Hello, World”.
We can also concatenate using the += operator. This is useful for when you are concatenating onto the same string over and over.

```
var myVar = “Hello my name is “;
myVar += “John ”;
myVar += “Doe “;
```

What does myVar now say? _____

We can also concatenate with variables.

```
var myAnimal = “cat “;
var name = “Johnny ”;
var myString = name + “likes to pet his “ + myAnimal;
```

Strings

The first function we will look at involving strings is the length function. This function will tell you the length of a string. We use it by attaching .length to end of a string or variable that contains a string.

Example

```
var myPet = “Cat”;
var myPetLength = myPet.length;
```

Bracket Notation

So let’s say you as a programmer would like to get a certain character out of a string. This is where bracket notation comes in handy. The first thing to understand is that JavaScript starts counting at 0 and not 1. This is known as Zero-based indexing.

Example

```
var myString = “Cat”;
var myChar = myString[0];
```

In this example, we are assigning the variable myChar to be the character located at the zeroth index of myString. This means myChar will be “C”.

Example

```
var myString = “The Cat in the Hat”;
var char1 = myString[0]; \\ char1 = T
var char2 = myString[5]; \\ char2 = a
var char3 =myString[8]; \\ char3 = i
```

We can use a combination of the length function and subtraction operators to find the nth to last characters in a string.

The following code is showing us how to find the 2nd to last character in the variable firstName.

```
var firstName = “Johnny”
Var nthChar = firstName[firstName.length - 2];

/*let’s break down the code that was just written.
   firstName.length gives us the length of firstName. Which is 6
   We then subtract 2 from 6 to get the index of the 2nd to last character.
  This simplifies the code to look like this firstName[4]
   And the nthChar will be n */
```

This is a good time to introduce the concept of Immutability. What this means is that a string cannot be altered once created. So we can’t change the characters within a string unless we recreate the entire string.

This is not allowed

```
var myStr = “Bob”;
myStr[0] = “J”;
```

The code written above will not work because strings are immutable.

The only way to change the content of a variable is to reassign it. Like the following:

```
var myStr = “Bob”;
myStr = “Job”;
```

Arrays

Array variables allow you to store several pieces of data in one place. An array would look like the following:

```
var pets = ['Cat', 'Dog', 'Hamster']; // this is an array of strings
var num = [16, 18, 19]; // Arrays can also contain numbers
var ran = [13, 'books', 25]; // An array can also contain different types of data types
var empty = []; // This is an empty array
```

You can also nest an array within another array. The following array will contain arrays of the names and ages of individuals.

```
var names = [['John Doe', 25], ['Jane Doe', 26]];
```

Content within an array can be accessed using bracket notation. Very much like strings, arrays use zero-based indexing.

Example

```
// Let's create an array
var array  = [13, 50, 76, 110, 46];
// This array contains 5 numbers
var data1 = array[0];  // equals 13
var data2 = array[3];  // equals 110
```

Entries within an array are mutable, which means they can be changed freely.

Example

```
var array = [12,13,14];
array[0] = 15;
// This changes array to now equal [15,13,14]
```

We now know how to access the data within an array using bracket notation, but now let's take it a step further and access the data contained within a nested array.

Example

```
var array = [
[1,2,3],
[2,4,6],
[3,6,9],
[4,8,12],
[5,10,15]
];
// This array contains 5 different arrays with 3 numbers contained in each

array[2]; //This is telling us to go to the 2 index of the top most array. [3,6,9]
array[2][1];
/* The first bracket refers to the entries in the outermost array. Then the next bracket
```

```
refers to the next level of entries.

array[2] => [3,6,9]
Then we look at the 1st index of that array.

So array[2][1] = 6
*/
```

Manipulating Arrays

The following functions allow us to add and remove data from an array:

- push() -> Takes one or more parameters and pushes onto the end of the array.
- pop() -> Removes the last value of an array, that value can be assigned to another variable.
- shift() -> Works just like pop() but removes the value from the front of the array.
- unshift() -> Adds elements to the beginning of an array.

```
var myArray = [10, 13, 7];
// Push
myArray.push(6); // myArray => [10, 13, 7, 6]
// Pop
myArray.pop(); // myArray => [10,13,7]
// Shift
myArray.shift();  // myArray => [13,7]
//unshift
myArray.unshift(23);  // myArray => [23,13,7]
```

JavaScript Functions:

JavaScript allows us to divide our code up into reusable parts known as functions.

```
function functionName() {
   console.log("Hello World");
}
```

We can call upon this function by just writing the code

```
functionName();
```

And it will post Hello World to the console.

We can also pass parameters or values that are used by the function into our functions.

```
function testFun(param1, param2){
  console.log(param1, param2);
}
```

This function written above will post both the parameters to the console.

Example:

```
function area(length, width){
  var area = length*width;
  console.log(area);
}
// This function takes in a length and width and returns the area of a rectangle.
```

This brings us to a concept known as scope; variables defined outside of a function block are considered global and can be used throughout the code, while variables defined within a function only exist within that function.

If you accidentally make a variable that doesn't use the var keyword, that variable is automatically made global. This can cause issues within your code, especially if you would call the function again.

```
var global = “This is a Global Variable”; /* Being global means it can be used anywhere in the code */

function fun(){
  var str = “This variable only exist within this function”;
  /* the variable str is only visible within this function, and cannot be seen outside of it. */
  oops = “This has accidentally created a global variable”;
}
```

We can have situations where the local and global variables share a name. In that situation the local variable will take precedence.

Functions can also return a value when they are called. This is done using the return statement.

```
function area(length, width){
  var area = length * width;
  return area;
```

```
}

var answer = area(5,4); // This will give us an answer of 20
```

Functions do not require a return value, and in that case the return value would be undefined. A good example of this would be a function that modifies some kind of global variable.

```
var num = 0;
function addThree(){
   num = num  + 3;
}

/* This function modifies the global variable num, but does not return a value. As a result
a value of undefined will be returned. So if you try to use it like this
   var newNum = addThree();
Nothing is returned so the value of newNum would be undefined. */
```

Understanding Boolean Values

Boolean is a data type that is used in JavaScript. It can be either true or false. True and false will never have quotes around them.

Conditional Statements

If statements are used to make decisions within our code, so if a condition is met then the code will run. These conditions that are tested for a known as Boolean Conditions, and they can only be true or false.

```
If (condition is true) {
  Statement is executed
}
```

We have many different comparison operators in JavaScript that return a boolean true or false.

Equality Operator	==	Compares two values and returns true if equivalent. Performs a type conversion.
Strict Equality Operator	===	Compares two values, but does not perform a type

		conversion.
Inequality Operator	!=	(Not Equal) Returns false where equality would return true and vice versa. Will convert data types while comparing
Strict Inequality Operator	!==	(Strictly Not Equal) Returns false where strict equality would return true and vice versa. Will not convert data types.
Greater Than Operator	>	Compares two values, if the number of the left is greater, then it returns true. Otherwise it returns false.
Greater Than Equal To Operator	>=	If number to the left is greater than or equal to the right. Returns true.
Less Than Operator	<	If number to the left is less than the number on the right. Returns True
Less Than Equal To Operator	<=	If number to the left is less than or equal to the number to the right. Returns True

Sometimes you need to test more than one thing at a time. As a result, we have two different logical operators.

The And Operator (&&) will only return true if, and only if, the values on the left and right are true.

true && true => true
true && false => false
false && true => false
false && false => false

The Or operator (||) will only return true if either of the values is true.

true || true => true
true || false => true
false || true => true
false || false => false

Conditional Statements Continued

When an if statement condition is true, that block of code runs. When that condition is not met, then nothing normally happens. This introduces us to else statement, which is a block of code that will run if the if statement does not run.

```
if (num > 15) {
   return “The number is larger than 15”;
} else {
   return “15 or less”;
}

// If num is greater than 15 then the if statement is run, But if num is not greater than, then the else statement runs.
```

So what happens if you have multiple conditions that need to be addressed? We can chain if statements together with what is known as an else if statement.

```
if (num > 15) {
   return “Bigger than 15”;
} else if (num < 5) {
   return “smaller than 5”;
} else {
   return “Between 5 and 15”;
}

/* In the code above we have an if statement that checks to see if the number is greater than 15. If that condentional is not met we then check the next conditional. Which is to see if the number is less than 5. Then finally if neither conditional is met we use the else
```

```
statement. */
```

The order is important in if, else if statements. They run from top to bottom, so you want to be careful which statement comes first.

You can chain together if/else statements for complex logic.

```
if (condition) {
   Statement 1
} else if (condition 2){
   Statement 2
} else if (condition 3){
   Statement 3
} else {
   Last Statement
}
```

Switch Statements

Switch statements test a value and can have many case statements define various possible values. Statements are executed from the first matched case value until a break is encountered.

Case values are tested with strict equality (===)
Break tells javascript to stop executing statements, and if break is omitted, the next statement will be executed.

In a switch statement you may not be able to specify all possible values as case statements. So we can add the default statement, which will be executed if no matching statements are found.

```
switch (val) {
   case 1:
      Statement 1;
      Break;
   case 2:
      Statement 2;
      Break;
   . . . .
   default:
      defaultStatement;
      Break;
```

```
}
```

So a switch statement will check the value against different cases, then run a statement if that case is met. It will continue to go through each case until it reaches a break statement. If there is no case found, we can set a default case in which that statement will happen instead.

If the break statement is omitted from a case, then the following case statements are executed until a break is encountered.

```
switch(val) {
  case 1:
  case 2:
  case 3:
    str = "This includes 1, 2, and 3";
    break;
  case 4:
    str = "This will only be 4";
```

We can actually convert an if/else statement into a switch statement. So let's look at the following if/else statement.

```
if (val === 1) {
  answer = "a";
} else if (val === 2) {
  answer = "b";
} else {
  answer = "c";
}
```

This if/else statements check to see if val is equivalent to 1, then it checks to see if it is equivalent to 2. If val is not equivalent to neither of those numbers, we then go straight to the else statement. Let's turn this into a switch statement.

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

```
}
```

JavaScript Objects

Objects are similar to arrays, but instead of using indexes to access the data, we use properties. Objects allow us to store data in a structured way.

```
var cat = {
  "Name": "Pippen",
  "Breed": "Tabby",
  "Enemies": ["Water", "Dogs"],
  "Favorite Place": "Owners Bed"
};
```

This object that was created above has the properties of Name, Breed, Enemies, and Favorite Place.

We can access the data within an object using dot notation. This requires us to know the name of the properties within the object. So let's say we want to know the breed of the cat object we created above.

```
var catBreed = cat.Breed // This will set catBreed to be Tabby.
```

If we have a property that contains a space in the name, we have to use bracket notation. Bracket notation can be used with properties that don't have a space in the name, but it is required for properties with a space in the name.

```
var favPlace = cat["Favorite Place"]; // This will set favPlace to Owners Bed
```

Let's look at our cat object one more time.

```
var cat = {
  "Name": "Pippen",
  "Breed": "Tabby",
};
```

So let's say that we wish to change the cat's name down the road. We can do that using dot or bracket notation.

```
cat.Name = "Frodo"; // or
```

```
cat[“Name”] = “Frodo”;
```

Using this notation allows us to update the content within our properties. This same notation allows us to also add new properties to our object. So if we wanted to add an age property, we would run code along these lines.

```
cat.Age = 2; // or
cat[“Age”] = 2;
```

Now our cat object looks like this.

```
var cat = {
   “Name”: “Frodo”,
   “Breed”: “Tabby”,
   “Age”: 2
};
```

Not only can we add properties, we can delete them.

```
delete cat.Age;
// This line of code will delete the Age property from the cat object.
```

If we have tabular data, we can use an object to lookup the values instead of using a switch or if/else statement.

The following object contains countries and their abbreviations. So we can use the abbreviation to find out which country it goes to.

```
var countries = {
   “usa”: “United States”,
   “ca”: “Canada”,
   “de”: “Germany”,
   “uk”: “United Kingdom”
};
```

We have this object and it allows us to quickly search for countries by their abbreviation.

```
countries[“de”]; // This will return Germany
```

If we have an object and we are unsure if it contains a certain property or not, we can use the hasOwnProperty method. This method will return true or false if the property is found or not.

```
var cat = {
  name:"Pippen",
  Age:2
};

cat.hasOwnProperty("name"); // This will return true
cat.hasOwnProperty("breed") // this will return false
```

Iterating with loops

Loops will continue to run as a specified condition is true, but once that condition is no longer true, the while loop will quit running.

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

Let's, breakdown the while-loop we just wrote. First, we have a variable of i that equals 0. The while loop will continue to iterate while i is less than 5. Each iteration will push that variable into an array and then add 1 to the value of i. This will continue until i greater than 5.

Now we have for loops, and they run for a specific number of times. They are declared with three optional expressions separated by semicolons.

for ([initialization]; [condition]; [final-expression])

- The initialization statement is executed one time only.
- The condition statement is evaluated at the beginning of every loop iteration and will continue as long as it evaluates.
- The final expression is the last thing the loop does; it is usually to increment or decrement your loop counter.

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

Looking at the for loop above, we see that variable i is initialized. While i is less than 5, we will push i to the array ourArray. Finally, 1 is added to i. This will continue as long as i less than 5.

```
var Array = [];
for (var i = 10; i > 0; i--){
   Array.push(i);
}
```

This for loop will give i a value of 10, and will continue to push the value of i into the array while i is greater than 0. After each iteration i, will decrease by 1.

You can also use a for loop to go through an array.

```
var arr = [10,25,13];
for (var i = 0; i < arr.length; i++){
   console.log(arr[i]);
}

// We start off with i being set to 0.
//Then while i less than the length of the array. Each index in the array will be printed to the console.
```

The final type of loop that is used is the Do While loop. This loop will do one pass of the code inside the loop no matter what. Then it will continue to run as long as the condition is true.

```
var arr = [];
var i = 0;
do {
   arr.push(i);  // The do will run at least once even if the condition is not met.
   I++;
} while ( i < 5) ; // While the condition is met the do will run again until the condition is no longer met.
```

Useful functions

Math.Random()	Generates a random number between 0 and not quite up to 1.

Math.floor()	Rounds the number down to the nearest whole number.
parseInt()	Parses a string and returns an integer

Ternary Operator

Allows us to create a one line if-else expression using the following format.

```
condition ? statement-if-true : statement-if-false;
```

Using ternary operator allows us to simplify if/else statements. Let's look at the next example:

```
function findGreater(a,b) {
   if(a > b) {
      return "a is greater";
   } else {
      return "b is greater";
   }
}
```

We can now rewrite that if/else statement using conditional operator:

```
function findGreater(a,b) {
   return a > b ? "a is greater" : "b is greater";
}
```

We can chain together Ternary Operators to check for multiple conditions. Let's look at the following if, else/if, else statement:

```
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";
  }
}
```

Now we will rewrite that function using multiple ternary operators:

```
function findGreaterOrEqual(a, b) {
  return (a === b) ? "a and b are equal" : (a > b) ? "a is greater" : "b is greater";
}
```

JavaScript Examples

If/Else Statements

If time is less than 10:00, create a "Good morning" greeting, if not, but time is less than 20:00, create a "Good day" greeting, otherwise a "Good evening":

```
if (time < 10) {
  greeting = "Good morning";
 } else if (time < 20) {
  greeting = "Good day";
 } else {
  greeting = "Good evening";
 }
```

If the grade is above 90 then give an A, if above an 80 give a B, if above a 70 a C, above a 60 a D and if below a 60 give a F.

```
if (grade < 0 || grade > 100){
  console.log("Invalid Grade")
} else if(grade >= 90){
  console.log("A");
} else if(grade >= 80) {
  console.log("B");
} else if(grade >= 70){
  console.log("C");
} else if(grade >= 60){
  console.log("D");
} else if(grade < 60){
  console.log("F");
}
```

Switch Statements

This example uses the weekday number to calculate the weekday name:

```
switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
    break;
  case 1:
    day = "Monday";
    break;
  case 2:
     day = "Tuesday";
    break;
  case 3:
    day = "Wednesday";
    break;
  case 4:
    day = "Thursday";
    break;
  case 5:
    day = "Friday";
    break;
  case 6:
    day = "Saturday";
}
```

For Loops

This will go through and list out every number from 0 to 4

```
for (i = 0; i < 5; i++) {
  console.log("The number is " + i);
}
```

This will go through a gradebook and list out the grade for each student.

```
let gradeBook = [["John",87],["Jill",95],["Blake",76],["Nate",54]];
```

```
for (let i = 0; i < gradeBook.length; i++){
  letterGrade = ''
  if(gradeBook[i][1] >= 90){
     letterGrade = 'A'
  }else if (gradeBook[i][1] >= 80){
     letterGrade = 'B'
  }else if(gradeBook[i][1] >= 70){
     letterGrade = 'C'
  }else if(gradeBook[i][1] >= 60){
     letterGrade = 'D'
  }else if(gradeBook[i][1] < 60){
     letterGrade = 'F'
  }
  console.log(gradeBook[i][0] + "'s grade is a " + letterGrade)
}
```

While Loops

```
let n = 0;
while(n < 3){
  console.log(n)
  n++
}
```

Functions

The Following function returns the sum of two parameters.

```
const sum = function(x,y){
  return x + y
}
```

The Next Function will take in a grade and then return a letter value.

```
const letterGrade = function(grade){

  if (grade < 0 || grade > 100){
```

```
    return "Invalid Grade"
  } else if(grade >= 90){
    return "A"
  } else if(grade >= 80) {
    return "B"
  } else if(grade >= 70){
    return "C"
  } else if(grade >= 60){
    return "D"
  } else if(grade < 60){
    return "F"
  } }
```

Basic JavaScript Review Questions

1. What is the remainder operator?

2. How do you declare a string variable?

3. What are the different data types that javaScript supports? (There are 7)

4. What are two different ways to access data contained within an object?

5. Give an example of a nested array?

6. What are the four methods to manipulate an array?

7. What are all the comparison operators?

8. Create a function that takes in a number and will add five to the number if the number is even.

9. Create a dog object that has 3 properties of your choice.

10. Arr = [1, 2, 4, 7, 11, 54, 88, 34]

Create a for loop that will return all even numbers in the array.

Bonus:

Make a ternary operator that returns a p tag that says Hello World if the condition is true and a p tag that says Goodbye if not true.

Introduction to the ES6 Challenges

ECMAScript is a standardized version of JavaScript with the goal of unifying the language's specifications and features. As all major browsers and JavaScript-runtimes follow this specification, the term *ECMAScript* is interchangeable with the term *JavaScript*.

Most of the challenges on freeCodeCamp use the ECMAScript 5 (ES5) specification of the language, finalized in 2009. But JavaScript is an evolving programming language. As features are added and revisions are made, new versions of the language are released for use by developers.

The most recent standardized version is called ECMAScript 6 (ES6), released in 2015. This new version of the language adds some powerful features that will be covered in this section of challenges, including:

- Arrow functions
- Classes
- Modules
- Promises
- Generators
- let and const

Note
Not all browsers support ES6 features. If you use ES6 in your own projects, you may need to use a program (transpiler) to convert your ES6 code into ES5 until browsers support ES6.

Var and Let

Var allows us to create a variable and we can overwrite it at any time. This can cause a problem in larger programs in which you might overwrite a variable you need. So in ES6

they introduced let. Using let helps us make sure we don't run into the issues of accidentally overwriting an existing variable. So a variable declared with let can only be declared once.

```
let camper = "James";
let camper = "David"; // This will throw an error.
```

ES6 also introduced "use strict". This enables strict mode, which catches common coding mistakes and unsafe actions. One example it would stop is the following:

```
"use strict"
x = 3.14;
// This would throw an error because the x is not declared.
```

The let keyword acts very similarly to the var keyword, but has extra features. For example, when you declare a variable with the let keyword within a block, statement, or expression, the scope of that variable is limited to that block, statement, or expression.

Const keyword

In ES6, we also have the const keyword. This keyword has all the features that let has, but these variables are read only. They are a constant variable, which means they can not be reassigned. Common Practice has us use all caps when declaring a const variable.

```
"use strict"
const FAV_PET = "Cats";
FAV_PET = "Dogs"; // This will return an error
```

Many Developers prefer to assign all their variables using const by default. But one good thing is that variables using const are still mutable. Using const only prevents reassignment of the variable identifier.

```
Const ARR = [1,2,3];
ARR = [2,4,6]; // This will throw an error
ARR[0] = 6; // Because variables created using const are mutable this is allowed
console.log(ARR); //This will return [6,2,3]
```

Preventing Object Mutation

Using const doesn't prevent your data from being mutated. So if you wish to make sure your data does not change, ES6 has introduced Object.freeze.

```
let obj = {
   name:"John Doe",
   Age:"25"
};
// We have created an object and now we will freeze it
Object.freeze(obj);
obj.age = 35; //This will be ignored
obj.newProp = "Test"; //Will be ignored

console.log(obj);
// { name: "John Doe",
       Age: "25" }
// As we see when we freeze an object it makes sure we can't mutate it.
```

Arrow Functions

In JavaScript, we don't always need to name our functions. The situation is seen when we are passing a function as an argument to another function. So we normally would create a variable with the function assigned to it, like the following:

```
const fun = function() {
   const var = "test";
   return var;
}
```

Instead of doing this, we can actually use an arrow for this function we are creating:

```
const fun = () => {
   const var = "test";
   return var;
}
```

But we can make this similar if the function we are writing only contains a return value. We can rewrite it as follows:

```
const fun = () => "test"
```

This code will return "test" when it runs.
We can also pass parameters into our arrow functions. The following code is a function to calculate the area of a rectangle:

```
const area = (length,width) => length*width;
```

Arrow Functions are really useful when used along with higher order functions. The following functions are examples of those:

map()	Creates a new array and calls the provided function once for each element in an array.
filter()	Creates an array filled with all array elements that pass a test, which is provided as a function.
reduce()	Reduces the array to a single value. It will execute a provided function for each value of the array.

Arrow functions are very useful to use along with these higher order functions. Let's look at an example:

```
const numbers = [5, 12, 20];l

const divByFive = numbers.filter( (num) => num % 5 === 0)

// So we are using, filter along with the arrow function that determines if a number is divisible by 5.
```

Default Parameters

ES6 introduced default parameters to JavaScript. What this means is that if no parameter is used, then a default parameter will be used instead.

```
function area(length = 1, width = 1){
   return length*width;
}

console.log(area(5,5));  // This will print an area of 25 to the console.
console.log(area()); // We did not include any parameters so the default ones of 1 will be used giving us an area of 1.
```

Rest Operator

ES6 has introduced the rest operator for function parameters. What this means is that you can create functions that take any number of arguments.

```
function fun(...args) {
   return args.length;
}

// You can input as many parameters as needed and all this function will do is return the number of parameters you have entered.
```

Spread Operator

Spread Operators allow us to expand arrays and other expressions in places where multiple parameters or elements are expected.

```
const arr = [6, 89, 3, 45];
const maximus = Math.max(...arr); // returns 89
// using the spread operator allows us to unpack the entire array so we can easily input an array as a parameter within a function.
```

Destructuring Assignment

Destructuring assignment is a special syntax that allows us to assign values taken directly from an object.

```
var cat= {
   name : “Pippin”,
   age : 18,
   breed: “Calico”
}

// We normally assign the values like this
var name = cat.name;
var age = cat.age;
var breed = cat.breed;

//But we can use Destructuring assignments to make it simple

const {name, age, breed} = cat;
// This creates 3 variables with the following values. name = pippin, age = 18, breed = calico

//What if we want to take one value from the object or choose which gets saved to what.
```

```
const { name: catName, age:catAge } = cat;

//This will take the property name and age, and save them to the variables catName and catAge.
```

We can also use destructuring to assign variables from nested objects.

```
const cats = {
   names : {name1:"Pippin", name2:"Frodo},
   age : {age1: 18, age2:10}
}

// now let's say we want the age of the 1st cat saved to a variable.

const {age : { age1 : catOneAge }} = cats

// This will create a variable called catOneAge and the assigned variable is 18.
```

We can also use destructuring to assign variables from arrays.

```
//We start at index 0 and move forward with each new variable we add.
const [a, b] = [1,2,3,4,5]
// a = 1 and b = 2

//We can also choose which index we want to use by adding commas to reach the desired index.

const [a, b, ,,c] = [1,2,3,4,5]
// a = 1, b = 2, c = 5  The two commas before c makes sure we skip 2 indexes. So we end up at 5.
```

We can also use destructuring to assign the remaining variables to a seperate array.

```
const [a, b, ...arr] = [1,2,3,4,5];
console.log(a,b) // This shows us 1 and 2
console.log(arr) // This shows us the array [3,4,5]
```

Strings using Template Literals

We can create more complex strings by using what is known as the string interpolation feature.

```
const cat = {
```

```
  name : “Pippin”,
  age : 5
};

const str = ‘Hello, my cat’s name is ${cat.name} and is ${cat.age} years old.’

// This produces a string that says the following.
// Hello ,my cat’s name is Pippin and is 5 years old.
```

Concise Object Literal Declarations

```
// The following function creates a new cat object.
const makeCat = (name,age) => ({
  name:name,
  age:age
});

// Using ES6 we can simplify it so it looks more like the following.

const makeCat = (name,age) => ({name,age});
```

Concise Declarative Functions

Before, we used the keyword function when defining a function within an object. In ES6 you can remove the function keyword. Let’s look at the following examples:

```
const cat = {
  name : “Pippin”,
  sayMeow: function() {
    return “Meow”;
  }
};

// This is how a function would normally be applied within an object.
// How with ES6 we can create a function like the following.
const cat = {
  name : “Pippin”,
  sayMeow() {
    return “Meow”;
  }
};
```

Class Syntax to Define Constructor Functions

ES6 has provided a new syntax to help create objects by using the keyword class. First, let's describe a constructor method. It is a special method for creating and initializing an object created within a class.

```
class Polygon {
  constructor(polygonType){
    this.polygonType = polygonType;
  }
}

const triangle = new Polygon('Triangle');
```

Getters and Setters

Using getters and setters, we can obtain values from an object and a set a value of a property within an object. Getters are meant to return the value of an object's private variable, while setters are meant to modify the value.

```
class car {
  constructor(modell) {
    this.model = model;
  }

  get carType() {
    return this.model;
  }

  set carType(updatedModel) {
    this.model  = updatedModel;
  }
};

const newCar = new Car('Ranger');
// Creates a new Car with a model of Ranger

console.log(newCar.carType);
// This will post ranger, because the current model is ranger
newCar.carType = 'Altima';
//We have just set the model to Altima

console.log(newCar.carType);
//This will print the updated car type.
```

Importing Functions

In order to import a function to a JavaScript file, you want to use the following syntax:

```
import { function } from "file path"
// You can also import variables
```

So let's say we want to use countItems from the math_array_functions package.

```
import { countItems } from "math_array_functions"
```

In order to import a function or variable, we first need to export it from where it originates. Once a piece of code is exported, we can import it anywhere we need it.

```
const area = (length,width) => { return length*width; }
// if we want to export this function we just use the following code.
export {area}
// we can also export variables
export const length = 10;


// We can export each object separately or compact all of them into one statement

export {area, length}
```

Let's say we want to import all of the functions within a package. We would use the following type of syntax:

```
import * as name_of_your_choice from "file path"
// So let's use it

import * as mathStuff from "math_functions";

// and if we wish to use it

mathStuff.add(2,1);
mathStuff.subtract(5,2);
```

We also have another type of export known as the export default. It is used if one value is being exported, or if a fallback value is needed. It is important to note that exported default can not be used with var, let, or const variables.

```
export default function mult(x,y) {
   return x * y;
}
```

When a file has a default export, you need to be able to import that export. Above we created a default export function called mult. We will now import that function.

```
import mult from “math_functions”;

mult(5,3); // Gives us 15
```

Basic ES6 Review Questions

1. What is ES6?

2. What is the difference between var and let?

3. What is the const keyword?

4. Convert this function to an arrow function.

```
const fun = function( ) {
    Const var = “test”;
    return var;
}
```

5. What is a default parameter?

6. What is the rest operator?

7. How can you copy an array? (use one of the operator’s)

8. Import a function from file path.

9. How can you prevent object mutation?

10. How many times can a variable be declared with let?

Introduction to the Regular Expression Challenges

Regular expressions are special strings that represent a search pattern. Also known as "regex" or "regexp", they help programmers match, search, and replace text. Regular expressions can appear cryptic because a few characters have special meaning. The goal

is to combine the symbols and text into a pattern that matches what you want, but only what you want. This section will cover the characters, a few shortcuts, and the common uses for writing regular expressions.

Test Method

The first method we will be working on is the test method. This method will test to see if that pattern we are looking for exists within the string.

```
let myStr = "The Cat in the Hat"; // We have created a string
let myRegex = /The/; // This is a regex, it is the pattern we are looking for. It must between / /
myRegex.test(myStr); // This will return true
```

Something important to note though is that the regex must be a literal match for what we are searching.

If we wish to search for multiple patterns, we use the or | operator.

```
let regex = /tea|soda|coffee/;
// This regex has three different patterns that can be searched for.
```

Earlier, it was mentioned that the pattern that will be searched for must be a literal copy of what's inputted. We can actually get around that by using the i flag. This flag tells the regex that any variation of that word can be searched for.

```
let str = "The Cat in the hat";
let catRegex = /CAT/i;
let result = catRegex.test(str);
// Using the i flag this will return true.
```

Extracting Matches

Along with testing to see if a pattern exists or not, we can extract the actual match we found by using the .match() method.

```
let str = "The Cat in the Hat";
let regex = "Cat";
str.match(regex);
```

```
// This will return ["Cat"]
```

Another flag we can use is the g flag. This flag will search or extract from a pattern more than once. And if you want to compound it with another flag, just use it like the following:

```
let str = "The Cat in the Hat";
let regex = /the/ig
str.match(regex);
// This will return both ["The", "the"]
// It will search multiple times with the g flag, and i flag ignores cases.
```

In a situation where you want to extract all of the words that begin or end with a certain letter, we can use the wildcard period.

```
let catStr = "The Cat in the Hat";
let regex = /.at/g
catStr.match(regex);
// This will return ["Cat","Hat"];
// We used the wildcard period  followed by at to say we want all words that end with at and then used the g flag to search through multiple times.
```

We can also search for a literal pattern with some flexibility with character classes. We can define a group of characters we wish to match by placing them inside square brackets. [ ]

So let's say we want to match "Run", "Ron", and we don't want to match "Ran". We would create a regex that looks like the following: /R[uo]n/. This will make sure that we only match the characters "u" or "o".

```
let run = "Run";
let ron = "Ron";
let ran = "Ran";

let regex = /R[uo]n/;

run.match(regex) // returns ["Run"]
ron.match(regex) // returns ["R[on"]
ran.match(regex) // returns null
```

We can also define the range of characters to match using the hyphen character.

```
let str = “The Cat in the Hat likes to sleep on a mat”;
let regex = /[c-h]at/ig;
str.match(regex) // This will return [“Cat”, “Hat”]
```

We can also search for numbers using the hyphen, like the following:

```
let str = “Waffles011425”
let regex = /[a-z0-9]/ig’
str.match(regex);

// This will return [“Waffles011425”];
```

We can also match all negated characters using the caret character ^. For example, let’s search for every character that is not a vowel:

```
let str = “The Cat in the Hat”;
let regex = /[^aeiou]/ig

str.match(regex); // This will return [T,h, ,C,t, ,n, ,t,h, ,H,t]
```

Sometimes we need to match a character that appears one or more times in a row. In that situation, we will use the + character.

```
let str = “aaabbcc”;
let regex = /a+/g;
str.match(regex); //This will return [“aaa”];
```

The next character that you will learn is the * character. This character will match characters that occur zero or more times.

```
let str1 = “Yaaaaaaaaaaay”;
let str2 = “Yabbering”;
let str3 = “Yourself”;

let regex = /ya*/ig

str1.match(regex) // returns [“Yaaaaaaaaaaa”]
str2.match(regex) // returns [“Ya”]
```

```
str3.match(regex) // returns ["Y"]
```

There are two types of matching, greedy and lazy matching. Greedy matching finds the longest part of the string that fits the regex pattern and returns it as a match, while lazy matching will find the smallest possible part of the string that meets the regex pattern.

In order to use lazy matching you need to use the ? character to change it to lazy matching.

```
let str = "Titanic";
let regex = /t[a-z]*?i/

str.match(regex) // returns ["ti"];
```

String Patterns

We saw earlier that the caret character inside a character set is used to create a negated character set. However, outside of the character set, it is used to search for patterns at the beginning of strings.

```
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
```

If we want to search the end of strings, we need to use the dollar sign character $ at the end of the regex.

```
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
```

Quantity Specifiers

You can specify the lower and upper number of patterns with quantity specifiers. We use curly brackets. { }. We put two numbers between the curly brackets. For instance, to match the letter a appearing between 3 and 5 times in “ah”, the regex would be /a{3,5}h/

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

Sometimes you want to specify only the lower number of patterns with no upper limit. In that situation, we would not include an upper limit, like in the following code:

```
let str1 = “Noooooooooope”;
let str2 = “Noope”;

Let regex = /No{3,}p/;

regex.test(str1) // returns true because the o appears 3 or
more times.
regex.test(str2) // returns false because o only appears 2
times.
```

You can also just specify the number of matches just by placing one number by itself within the curly brackets, as in the following:

```
let str1 = “yaaay”;
let str2 = “yay”;

let regex = /ya{3}y/

regex.test(str1) // returns true
regex.test(str2) //returns false
```

In some situations, you can have a pattern that has parts of it that may or may not exist. For instance, you may have a character in the pattern that may not be there normally. A

good example is how some words are spelled slightly differenty in the UK than here in the United States. In order to check for this element you will use the ? character.

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

Lookaheads

Lookaheads are patterns that tell JavaScript to look ahead in your string to check for patterns further along. We have two types of lookaheads: Positive and Negative Lookaheads.

Positive Lookaheads - Look to make sure the element in the search pattern is there. They will not match it though. In order to use a positive lookahead, you need to use this syntax: (?=...) where the ... is the pattern that is not going to be matched.

Negative Lookaheads - Look to make sure the element in the search pattern is not there. It will use the following syntax. (?!...) where ... is the pattern that you do not want to be there.

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

Capture Groups

Some patterns will occur multiple times in a string, and it's wasteful to manually repeat that regex. This brings us to using capture groups. Parentheses are used to find repeated substrings. You will put the regex of the pattern that will repeat between them. Then to specify where that repeat string will appear, you will need to use a backslash(\) and a number. The number will start at 1 and increase with each capture group that is used.

```
let repeatStr = "regex regex";
let repeatRegex = /(\w+)\s\1/;
```

```
repeatRegex.test(repeatStr);   // Returns true
repeatStr.match(repeatRegex);   // Returns ["regex regex",
"regex"]
```

Search and Replace

Regular expressions allow us to search and replace using the replace method. Let's look at an example:

```
let str = "The Dog in the Hat";
let regex = /Dog/;

str.replace(regex, "Cat");
// This will return "The Cat in the Hat"
```

Shortcut Chart

These are important shortcuts to know when dealing with regex expressions.

\w	Equivalent to [A-Za-z0-9] Matches everything
\W	This will match everything but letters and numbers.
\d	Matches all numbers [0-9]
\D	Matches all Non-Numbers
\s	Searches for Whitespace
\S	Matches Non-Whitespace
+	Matches the preceding expressing 1 or more times.
*	Matches the preceding expression 0 or more times.
?	Matches the preceding expression 0 or 1 time, that is preceding pattern is optional
^	Matches beginning of the string

$	Matches end of the string
{N}	Matches at least N occurrences of the preceding regular expression, N is any number
{N,M}	Matches at least N to at most M occurrences of the preceding regular expression. M > N
X|Y	Matches either X or Y
(x)	Matches x and remembers the match. Capturing group
(?:x)	Matches x and does not remember the match.
x(?=y)	Matches x only if x is followed by y. Positive lookahead.
[^xyz]	Matches anything not enclosed within the brackets.

Examples

Extracting a number from a string

```
'Test 123456789'.match(/\d+/);

// Returns an array of ["123456789"]
```

Match a date with the following format DD-MM-YYYY or DD-MM-YY

```
var regex = /^(\d{1,2}-){2}\d{2}(\d{2})?$/;

regex.test('01-01-1990') // true

regex.test('01-01-90') // true

regex.test('01-01-190;) // false
```

Breaking down the problem

1. Using ^ to represent it begins with the first subexpression
2. (starts off the first subexpression
3. \d{1,2} We want 1 digit to 2 digits

4. - literal hyphen
5.) end of subexpression
6. {2} Match the first subexpression exactly 2 times.
7. \d{2} match exactly 2 digits
8. (\d{2})? Matches another 2 digits but its optional, meaning the year is 2 or 4 digits.
9. $ meaning the last subexpression is at the end only.

Checking for usernames

The rules for usernames:

1. Numbers at the end, Zero or more of them
2. Letters can be lowercase or uppercase
3. Usernames must be 2 characters long. A 2 character username must use only alphabet characters

```
let usercheck = /^[a-z]{2,}\d*$/i

let uname = 'Oceans 11'

let result = usercheck.test(uname) // returns true
```

Breakdown

1. ^[a-z] matches all letters in the beginning
2. {2,} The preceding match is matched 2 or more times.
3. \d*$ Matches zero or more digits at the end.

Basic Regular Expressions Review Questions

1. What are Regular Expressions?

2. How can you test if a pattern exists?

3. To search or extract a pattern more than once, you can use what flag?

4. What is the wildcard character?

5. How can you match characters that occur zero or more times?

6. How can you match characters that occur one or more times?

7. What is the shortcut to look for digit characters?

8. What is the shortcut to look for non-digit characters?

9. How can you specify the possible existence of an element with what mark?

10. What method would you use to remove all whitespace from start to end?

Introduction to the Debugging Challenges

Debugging is a valuable and (unfortunately) necessary tool for programmers. It follows the testing phase of checking if your code works as intended and discovering it does not. Debugging is the process of finding exactly what isn't working and fixing it. After spending time creating a brilliant block of code, it is tough realizing it may have errors. These issues generally come in three forms: 1) syntax errors that prevent a program from running, 2) runtime errors when code fails to execute or has unexpected behavior, and 3) semantic (or logical) errors when code doesn't do what it's meant to.

Modern code editors (and experience) can help identify syntax errors. Semantic and runtime errors are harder to find. They may cause your program to crash, make it run forever, or give incorrect output. Think of debugging as trying to understand why your code is behaving the way it is.

Example of a syntax error - often detected by the code editor:

```
function willNotWork( {
 console.log("Yuck");
}
// "function" keyword is misspelled and there's a missing parenthesis
```

Here's an example of a runtime error - often detected while the program executes:

```
function loopy() {
 while(true) {
   console.log("Hello, world!");
 }
}
// Calling loopy starts an infinite loop, which may crash your browser
```

Example of a semantic error - often detected after testing code output:

```
function calcAreaOfRect(w, h) {
 return w + h; // This should be w * h
}
let myRectArea = calcAreaOfRect(2, 3);
// Correct syntax and the program executes, but this gives the wrong answer.
```

Debugging is frustrating, but it helps to develop (and follow) a step-by-step approach to review your code. This means checking the intermediate values and types of variables to see if they are what they should be. You can start with a simple process of elimination.

For example, if function A works and returns what it's supposed to, then function B may have the issue. Or start checking values in a block of code from the middle to try to cut the search space in half. A problem in one spot indicates a bug in the first half of the code. If not, it's likely in the second.

This section will cover a couple of helpful tools to find bugs, and some of the common forms they take. Fortunately, debugging is a learnable skill that just requires a little patience and practice to master.

Basic Debugging Review Questions

1. What is Debugging?

2. What are the three main reasons your code usually won't work?

3. What would you use in your console to check or follow your code?

4. What are semantic or logical errors?

5. What is it called when you crash your computer?

Introduction to the Basic Data Structure Challenges

Data can be stored and accessed in many different ways, both in Javascript and other languages. This section will teach you how to manipulate arrays, as well as access and copy the information within them. It will also teach you how to manipulate and access the data within Javascript objects, using both dot and bracket notation. When you're done with this section, you should understand the basic properties and differences between arrays and objects, as well as how to choose which to use for a given purpose.

Common Debugging Challenges

- Make Use of console.log() to print out variables and functions
- Make use of typeof to make sure you are dealing with data structure type you should be.
- Misspelled Variables and Functions
- Checking to see if Parentheses, Brackets, Braces, and Quotes are all closed.
- Having mixed usage of single and double quotes
    - When working with strings make sure you stick with one or the other.

- Accidentally using the assignment Operator instead of the Equality Operator.
    - Example. If (x = 7) // That will not run'
- Forgetting to include your parenthesis after a function call.

```
function hello() {
  Return "Hello World";
}

let var = hello; // In this situation the parenthesis are not being called so it will not work properly.
```

- Making sure the arguments being passed into your functions are passed in the right order.
- Off by One Errors

- Happens when you are trying to target a certain index of a string or an array, or when looping over one.
- JavaScript indexing starts at 0, 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 you will get an index out of range error.

```
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]);
}
// by using less than equals, this loop will actually include the length of the array as well.
```

- Reinitializing variables inside a loop can lead to bugs as well, while sometimes it's necessary to save the information. The issue can arise when variables should be reinitialized and are not.
    - This can lead to infinite loop.
    - Using console.log() with variables in your loop can help uncover any issues.

Introduction to the Object Oriented Programming Challenges

At its core, software development solves a problem or achieves a result with computation. The software development process first defines a problem, then presents a solution. Object oriented programming is one of several major approaches to the software development process.

As its name implies, object oriented programming organizes code into object definitions. These are sometimes called classes, and they group together data with related behavior. The data is an object's *attributes*, and the behavior (or functions) are *methods*.

The object structure makes it flexible within a program. Objects can transfer information by calling and passing data to another object's methods. Also, new classes can receive, or inherit, all the features from a base or parent class. This helps to reduce repeated code.

Your choice of programming approach depends on a few factors. These include the type of problem, as well as how you want to structure your data and algorithms. This section covers object oriented programming principles in JavaScript.

Creating An Object

We looked at objects earlier on in our handbook, but we are now going to take a more in-depth look into them and how they are useful. Like objects we see in the real world, objects we create in JavaScript have properties as well.

```
let cat = {
   name: “Pippin”,
   color: “Orange”
};

// This object is a cat, and this cat has a name and a fur color. Which are its properties.
```

If you remember from earlier, we can access the details within an object by using dot or bracket notation.

```
let cat = {
   name: “Pippin”,
   color: “Orange”
};

console.log(cat.name);  // Dot Notation is object.property
console.log(cat[‘color’]) // Bracket Notation is object[‘Property’]
```

Objects can also have a special property called a method. These methods are properties that are functions and allow us to add different behavior to the object.

```
let cat = {
   name: “Pippin”,
   color: “Orange”,
   meow: function(){
      return “Meow”;
   }
};

cat.meow()
// This will return Meow
```

We can also refer to properties within our objects within our methods. Let’s update the meow function so that the cat object can now say its name.

```
let cat = {
   name: “Pippin”,
   color: “Orange”
   sayHello: function(){
      return “Hello, my name is “ + this.name+ “.”;
   }
};

// Within this method we used the this keyword to refer to the name property. To simplify what's going on this refers to the object itself.
```

Constructor Functions

Constructors are functions that create new objects. They define properties and behaviors that belong to this object, kind of like a blueprint. When making a constructor, you must follow a few conventions:

- Defined with a capital name to distinguish them from other functions.
- Use the keyword this to set properties that they will create.
- Define properties and behaviors instead of returning a value.

```
function Cat(name,color,age) {
   this.name = name;
   this.color = color;
   this.age = age;
   //”this” inside the constructor always refer to the object being created.
}
```

Once we have the constructor function made, we can then use it to make objects of that type.

```
function Cat(name,color) {
   this.name = name;
   this.color = color;
}

let tabby = new Cat(“Pippin”,”Orange”)

// These constructors have been designed to take in arguments to fill out the properties of the object.
```

Whenever a constructor function creates a new object, that object is stated to be an instance of its constructor. We can use instanceOf to compare an object to a constructor. It will return true or false based upon whether the constructor created the object.

```
function Cat(name,color) {
   this.name = name;
   this.color = color;
}

let tabby = new Cat("Pippin","Orange")

tabby instanceOf Cat; // This will return true

let calico = {
   name: "Ash",
   color: "White/Orange"
};

calico instanceOf Cat // This will return false
```

When we make a constructor, we have the properties that we are defining. These are known as their own properties. That is because they are defined directly on the instance object. What that means is that every instance of a constructor will have its own separate copy of the properties.

Sometimes when you have a duplicate variable, you want to use a prototype. The prototype is an object that is shared among all instances of the constructor.

```
function Cat (name,breed){
   this.name = name;
   this.breed = breed;
}

// Now we may have a duplicate variable that is needed across all the constructors.

// This is when we would use prototype

Cat.prototype.numTails = 1;

// Now every instance of Cat constructor will have this property.
```

We now know that objects have two kinds of properties, own properties and prototype properties.

- Own properties are defined directly on the object instance itself.
- Prototype properties are defined on the prototype.

We also have the constructor property located on the object instances. This property is a reference to the constructor that created the instance. We can use the property to find out what kind of object it is.

```
let pippin = new Cat();

console.log(pippin.constructor === Cat);
// This will return true.
```

One issue that can arise is that the constructor property can be overwritten. As a result, it is better to use the instanceOf method to check the type of an object.

Adding Multiple Prototypes

When we want to add multiple properties we want to add them all at once using the following set up:

```
Cat.prototype = {
  numLegs: 2,
  meow: function() {
    console.log("Meow");
  }
};

//This has given every instance of the Cat constructor those properties to used.
```

But when you set the prototype manually, like we did above, the constructor property will be erased. To fix that problem, we also need to define the constructor property.

```
Cat.prototype = {
  constructor: Cat, //This defines the constructor
  numLegs: 4,
  meow: function() {
    console.log('Meow');
```

```
  }
};
```

Something to note is that objects will inherit their prototype from the constructor function that created it. To show the relationship between the prototype and object, we can use the isPrototypeOf method.

```
function Cat(name) {
  this.name = name;
}

Cat.prototype = {
  constructor: Cat,
  numLegs: 4,
  meow: function() {
    console.log('Meow');
  }
};

let tabby = new Cat("Pippin");

Cat.prototype.isPrototypeOf(tabby);

// This will return true.
```

All objects usually have a prototype. There are instances in which they do not, however. A prototype itself is an object, which means it can have its own prototype. So this leaves us with a chain. In my code example above, the Cat Object has a prototype and is an object. That prototype is known as the Object.prototype.

So let's look at some code:

```
Cat.prototype = {
  constructor: Cat,
  eat: function() {
    console.log("nom nom nom");
  }
}

Dog.prototype = {
```

```
   constructor: Dog,
    eat: function() {
       console.log("nom nom nom");
   }
}


// So let's look at the code, These two prototypes have repeated code. The eat function.
```

The eat function in both of these prototypes is repeated, and this leads us to introduce a concept called Don't Repeat Yourself (DRY). We can create a supertype or parent prototype called Animal.

```
function Animal () {};

Animal.prototype = {
   constructor: Animal,
   eat: function() {
      console.log("nom nom nom");
   }
};
```

Your animal prototype now contains the, eat function, so we can remove it from the Cat and Dog prototypes.

```
Cat.prototype = {
   constructor: Cat
};

Dog.prototype = {
   constructor: Dog
};
```

You just saw how to create a supertype function. Now we will look at how to reuse the methods inside new objects using inheritance.

```
function () { }
Animal.prototype.eat = function() {
   console.log('nom nom nom');
};
```

```
// First lets create an instance of Animal
// We have seen how to create a new instance of an object before. In this case we need to take another approach.

let animal = Object.create(Animal.prototype);

//Object.create(obj) creates a new object and sets obj as the prototype.
```

Let's create the prototypes we will use for the Cat and Dog by using inheritance .

```
Cat.prototype = Object.create(Animal.prototype);
Dog.prototype = Object.create(Animal.prototype);

// So when we create a new Cat object it will inherit all of the methods from the animal prototype.

let tabby = new Cat("Pippin");
let corgi = new Dog("Frodo");

tabby.eat () //This will print nom nom nom
```

We do have an issue though. When an object inherits a prototype from another object, it also inherits its constructor.

```
function Cat() { }
Cat.prototype = Object.create(Animal.prototype);
let tabby = new Cat();
tabby.constructor // This will return the Animal Constructor. We want it to be the Cat constructor.

Cat.prototype.constructor = Cat;

// This will now change the constructor to be Cat.
```

A constructor function that inherits its prototype object from a supertype constructor function can still have its own methods. So let's take a look at how this is possible.

```
function Animal() { }
Animal.prototype.eat = function() {
  console.log("nom nom nom");
```

```
};
function Cat() { }
Cat.prototype = Object.create(Animal.prototype);
Cat.prototype.constructor = Cat;

// Now let's add a new method to the Cat prototype.

Cat.prototype.meow = function() {
  console.log('Meow');
};

// So now all instances of Cat will have eat() and meow()
```

We have learned that an object can inherit its behavior from another object by cloning its prototype object. In some situations, we need to override one of the methods. To do that, simply add a method using the same name as the one we are trying to override to the prototype.

```
function Animal () { }
Animal.prototype.eat = function() {
  console.log("nom nom nom");
};

function Bird() { }
Bird.prototype = Object.create(Animal.prototype);

//At this point we have created a Bird constructor that will inherit all of the animals methods.
Bird.prototype.eat = function() {
  console.log("peck peck peck");
};

// This has overwritten the orginal eat method from the animal prototype.
```

Mixin to add Common Behavior Between Unrelated Objects

Often methods are shared through inheritance, but in some situations inheritance is not the best option. When we have unrelated objects, we want to use what is called mixins. A mixin allows other objects to use a collection of functions.

```
let pushMixin = function(obj) {
  obj.push = function() {
```

```
    console.log("Pushing something forward");
  }
};

// This has created a mixin.

let hobbit = {
  name: "Pippin"
}

let snowPlow = {
  age: 2
}

pushMixin(hobbit);
pushMixin(snowPlow);

// Both hobbit and snowPlow have been passed into the pushMixin. Which will assign them the push function.

hobbit.push() // prints pushing something forward
snowPlow.push() // prints pushing something forward.
```

Using Closure to Protect Properties within an Object from being Modified Externally.

In previous examples, our objects used public properties that can be edited outside of the object's definition. But there are some things within our object that we do not want to be changeable. The simplest way to make properties private is to create a variable within the constructor function. The reason we would want this is so that the property can only be accessed and changed by methods within the constructor function.

```
function Cat() {
  let hoursSlept = 10;  // This is a private property

  this.getHoursSlept = function() {
    return hoursSlept;
  };
}

// We have created a new constructor with a private property. This makes sure that the property cannot be changed publicly.

let tabby = new Cat();
tabby.getHoursSlept(); // This will return 10
```

Immediately Invoked Function Expression (IIFE)

In some situations we want to immediately use a function as soon as its declared. These functions do not have a name, and they are not stored in a variable. We use parentheses () at the end of the function so it can be used immediately.

```
( function () {
   console.log(“Meow”);
})();

// This is a function that executes immediately.
```

We use IIFE to often group related functionality into a single object or module. A good example would be mixins.

```
function glideMixin(obj) {
  obj.glide = function() {
    console.log("Gliding on the water");
  };
}
function flyMixin(obj) {
  obj.fly = function() {
    console.log("Flying, wooosh!");
  };
}
```

We can now use IIFE to group these mixins into a module.

```
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
```

We are using IIFE to return an object. This returned object contains all of the mixin behaviors as properties of the object. This allows us to use mixins in other parts of our code.

Basic Object Oriented Programming Review Questions

1. Define what Object Oriented Programming is?

2. Create a dog object with three properties.

3. Create a dog constructor and give it a woof function.

4. Now create a new dog object and give it the woof function.

5. When adding multiple properties what would you use?

6. Objects will inherit their prototype from what?

7. To show the relationship between the prototype and object, you can use the _____ method.

8. Can a prototype have its own prototype? If so, explain.

9. What does (DRY) mean?

10. If you have two objects with the same function what would you use to keep it from repeating itself?

Introduction to the Functional Programming Challenges

Functional programming is an approach to software development based around the evaluation of functions. Like mathematics, functions in programming map input to output, producing a result. You can combine basic functions in many ways to build more and more complex programs.

Functional programming follows a few core principles:

- Functions are independent from the state of the program or global variables. They only depend on the arguments passed into them to make a calculation.
- Functions try to limit any changes to the state of the program and avoid changes to the global objects holding data .
- Functions have minimal side effects in the program.

The functional programming software development approach breaks a program into small, testable parts. This section covers basic functional programming principles in JavaScript.

Learning Functional Programming

Functional Programming is a style of programming in which solutions are simple, isolated functions, without any side effects of the function scope.

INPUT -> PROCESS -> OUTPUT

Functional Programming Terminology

Callbacks	Functions that are slipped or passed into another function to decide the invocation of that function.
First Class Functions	Functions that can be assigned to a variable, passed into another function, or returned from another function just like any other normal value.
Higher Order Functions	value.
lambda	Functions that are passed in a function or returned from a function.

Hazards of Using Imperative Code

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 good example of imperative programming is the for loop. It gives exact directions to iterate over the indices of an array.

Functional programming is a form of declarative programming. You basically tell the computer what you want done by calling a method or function. JavaScript offers many predefined methods that handle common tasks. For instance, we can use the map method instead of a for loop. The map method handles the details of iterating over an array. One big advantage of this is that it can handle “Off By One Errors”.

Functional Programming - Avoiding Mutations and Side Effects

One of the core principles of functional programming is to not change things. When we make changes, it can lead to bugs. In functional programming, changing or altering things is called mutation, and the outcome is called a side effect. Ideally, a function should not cause any side effects. It should be a Pure Function.

Pass Arguments to Avoid External Dependence

Another principle of functional programming is to always declare your dependencies explicitly. So if the function you are writing depends on a variable or object being present, then pass that variable or object directly into the function.

```
let num = 3;

function multFive (num) {
   return num*3;
}
```

Important Functions

map()	Iterates over each item in array. Creates a new array after applying a callback function to every element.
filter()	Take a callback function and will apply it on each element of the array. If an element returns true based on the criteria in the callback function. Then it is included in the new array.

slice()	Returns a copy of certain elements of an array. Can take two arguments. 1st gives the index where to begin slice, 2nd is where to end the slice.
concat()	Called on an array and adds the second array onto the first. This returns a new array containing the concatenation.
reduce()	Used to reduce the array to a single value and execute a provided function for each value. Then returns the value of the function in an accumulator.
split()	Splits a string into an array of strings. It takes an argument for the delimiter. Which will be the character to break up the string or regular expression.
join()	Method that joins elements of an array together to create a string. Take a delimiter that is used to separate the array elements.
every()	Checks to see if every element passes a test. Returns a boolean value if it passes or not.
some()	Checks to see if any element passes a test. Returns true or false

Examples of methods that are commonly used in functional programming.

```
const num = [2,4,8,10];
const halves = numpers.map(x => x/2);

// Using map is very similar to using a for loop. It will apply the function given to each element in the array.
```

```
const words = ['spay','limit','elite','exuberant','destruction',present'];

const longWords = words.filter(word => word.length > 6);

// Using the filter applied the function checking to see if the elements in word are greater than 6 characters.
```

```
const total = [0,1,2,3].reduce((sum,value) => sum + value,1);
```

```
// The reduce method applies a function against an accumulator. So it can reduce it to a single value.
```

```
var arr = [23,56,87,32,75,13];
arr.slice() // Returns an exact copy of the original array
arr.slice(2) // extracts the array starting from given index
arr.slice(2,3) //extracts the array starting from given index and includes all elements less than the second index.
```

```
let arr = [1,2,3];
arr.push(4);
// arr has been changed to [1,2,3,4].
// To use functional programming we should do the following.

let newArr = arr.concat(5);
// this creates a new array without modifying the original one.
```

```
let fruits = ['Banana', 'Orange', 'Apple', 'Mango'];
let energy = fruits.join( );
// energy will now be 'Banana Orange Apple Mango'

let string = 'The Cat in the Hat';
let arr = string.split( );
// split will spit the string on the given delimiter. In this case it is a white space.
// So arr = ['The','Cat','in','the','Hat'];
```

```
let arr = [1,30,39,29,10,13];
let bool = arr.every((currentValue) => {return currentValue < 40;})
// This will return true, because they are all lower than 40.
```

```
let arr = [1,2,3,4,5];
let bool = arr.some((num) => {return num%2 === 0;});

// This will return true because at least 2 of the elements are divisible by 2.
```

Basic Functional Programming Review Questions

1. What is Functional Programming?

2. Why is functional programming so important?

3. What are callback functions?

4. What are higher order functions?

5. What are the nine important functions? (Most commonly used)

Let arr = [1, 2, 3, 4, 5, 6, 7]

6. Use the array above and functional programming to return and arr with 1-8, without modifying the original array.

7. Use the array above and functional programming to return another array that has all of the even numbers, without modifying the original array.

8. Use functional programming to multiply each number by 2.

9. Use the array above and functional programming to return another array that has all of the odd numbers, without modifying the original array.

10. Use the array above and functional programming to return another array that has only 5, 6 and 7 in it, without modifying the original array.

Introduction to the JavaScript Algorithms and Data Structures Projects

Time to put your new JavaScript skills to work! These challenges will be similar to the algorithm scripting challenges but more difficult. This will allow you to prove how much you have learned.

In this section you will create the following small JavaScript programs:

- Palindrome Checker
- Roman Numeral Converter
- Caesars Cipher
- Telephone Number Validator
- Cash Register

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.
.
Note:
You'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".

Roman Numeral Converter

Convert the given number into a roman numeral.

All roman numerals answers should be provided in upper-case.

Note:
The principles/Rules of Roman numerals

1. Write numerals left to right, with the largest numeral first.
2. The largest numeral possible is used at each stage.
3. No more than three instances of the same adjacent numeral. Occasionally number 4 is written not as IV but as IIII to add symmetry and balance to a watch or clock face.
4. A smaller numeral such as I or X placed before a larger one has the effect of minus - thus IV is one less then five, or four. This is called the subtraction principle and only one numeral can be placed to the left. The small numeral must be a power of ten: I, X or C; (1, 10 or 100)

Caesar 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.

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) 555-5555
555 555 5555
5555555555
1 555 555 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.

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.

cidis a 2D array listing available currency.

The checkCashRegister()function should always return an object with a statuskey 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 Unit	Amount
Penny	$0.01
Nickel	$0.05
Dime	$0.10
Quarter	$0.25
Dollar	$1
Five Dollars	$5
Ten Dollars	$10
Twenty Dollars	$20
One Hundred Dollars	$100

When you feel like you are almost there, try one of these algorithms and turn it into your instructor. ***"GOOD LUCK!"***