






JavaScript

 Author	 Jay Vachhani
 Category	JS
 Date Started	@Jun 1, 2020
 Last Revision	@Jun 23, 2020

Acknowledgement:

I would like to personally thank all of the open source community in their continued effort to invent and innovate for the greater good of technology.

I have used multiple sources in writing this document. From online courses/articles/websites to also my own personal learning and knowledge. By no means am I claiming to be the sole creator of all of the information displayed here, It is a collection of curated material from myself and the wider web. I have written this document out myself, should you feel there are any major inconsistencies or inaccurate information being presented here, please contact me using the tab "Contact Me" on my website jayvachhani.tech , please use the name of this document in your message.

JavaScript

JavaScript is a high level programming language that all modern web browsers support. It is utilised in nearly all forms of web applications with a vast amount of libraries and frameworks to its disposal.

Webpages are split into 3 different layers:

1. HTML Markup → Content Layer, the actual content you want to display
2. CSS Rules → Presentation Layer, instructions on how to display HTML code
3. Javascript → Interactive Layer which can manipulate CSS and HTML

JavaScript & Java

These are unrelated languages. Java is used for applications for devices, JavaScript is a scripting language for interactive websites mainly.

The Ecosystem

JavaScript adheres to the ever-evolving ECMAScript specifications. ECMAScript is not a language, but rather a set of standards that browsers use to interpret JavaScript. Different browsers adhere to different revisions of ECMAScript, however most are similar and functionality of JS performs uniformly across these browsers. We will go into detail about ECMAScript and what newer versions of ECMAScript allow us to do with our programs.

Libraries

jQuery → functions that introduce CSS-like syntax and visual/UI enhancements

Libraries are an abstraction of the language and allow developers to code more efficiently.

Frameworks

AngularJS, React, Vue.js are front end development frameworks which are utilised in building advanced interactive web applications. They are used to simplify and standardise a web-application's building methodology and structure for programmers.

Platforms/Runtime Environments

These tools allow us to run advanced operations on applications on servers and computers using JavaScript. Node.js is an example of a platform built on browser runtime environments.

Key

```
// This is an inline comment
/* This is a
multi-line comment */

"use strict"; // this enables strict mode which detects coding errors
```

Data Types

JS provides 7 different data types:

undefined null boolean string symbol number object

Variables are used to store and manipulate these data types. You don't need to mention exact data types when creating variables like in Java or Python. Variable declaration and assignment looks like:

```
var myName = "Jay";
var myNum = 7;
var myOtherNum = 8;
myNum = myOtherNum; // This makes myNum's value become 8.
var a; // This variable is unassigned.
var myDecimal = 5.7; // no special declaration for floating point numbers

typeof var // returns the data type of the variable
```

All variable and functions are case-sensitive:

MYVAR \neq MyVar \neq myvar

Best practice is to write variable names in camelCase

Scope

Scope refers to the visibility of variables. variables declared outside of the function have Global Scope, not using var automatically puts them into the global scope.

It is possible to have both local and global variables with the same name, if this is the case then the local variable will take precedence.

Fundamental Mathematical Operations

Integer

```
var sum = 10 + 10; // sum = 20
var difference = 45 - 33; // difference = 12
var multiply = 8 * 10 ; // multiply = 80
var quotient = 66 / 33; // quotient = 2

// Increment by 1
var myVar = 1;
++myVar; // myVar = 1 + 1 = 2
myVar++; // myVar = 1 + 1 + 1 = 3

// Increment >1
var myVar = 1;
myVar += 3; // myVar = 1 + 3 = 4

//Decrement by 1
var myVar = 3
--myVar; // myVar = 3 - 1 = 2
myVar--; // myVar = 3 - 1 - 1 = 1

// Decrement >1
var myVar = 3;
myVar -= 2; // myVar = 3 - 2 = 1

// Multiplication
var myVar = 1;
myVar *= 5; // myVar = 1 x 5;
```

```
// Division
var myVar = 6;
myVar /= 3; // myVar = 6 / 3;

// Random decimal number between 0 and 1 (excluding 1, including 0)
Math.random()

// Random whole number between a range
Math.floor(Math.random() * (max - min + 1)) + min;

// Rounding a decimal down to nearest whole number
Math.floor

// Use parseInt() to convert String into Integer
var a = parseInt("007"); // a = 7
var b = parseInt("11", 2); // b = 3 , radix is used to tell which base to covert to
```

Decimal

```
var sum = 1.0 + 1.0; // sum = 2
var difference = 4.5 - 3.3; // difference = 1.2
var multiply = 2.0 * 2.5 ; // multiply = 5
var quotient = 4.4 / 2.0; // quotient = 2.2

// Remainder
var myRem = 5 % 2; // 5/2 has remainder 1, so myRem = 1
```

String Operations

```
// Escaping Literal Quotes in JS
var myStr = "Owen said, \"Ned has a big brain\".";
var myStr = 'Owen said, "Ned has a big brain".';
// Owen said, "Ned has a big brain".

// Concatenation
var myStr = "First part" + " and the Second part" // First part and the Second part
var combineStr = "First part";
combineStr += " and the Second Part"; // combineStr = myStr

var partStr = "fruit";
var fullStr = "Apple is a " + partStr; // Apple is a fruit
```

Single and Double quotes perform the same way in JS. A String must begin and end with the same type of quote. if there are the same quote in the text as a string, use `"\"` to resolve the bug.

Strings are immutable, cannot change individual characters, but can change the entire string.

Escape Sequences in Strings

```
"\'" = ' // single quote
"\"" = " // double quote
"\" = \ // back slash
"\n" = // newline
"\r" = // carriage return
"\t" = // tab
"\b" = // word boundary
"\f" = // form feed
```

Fundamental Data Structures

Arrays

```
// Declare an Array
var arr = [];
var arr = ["String", 2]; // Arrays can contain both strings and ints

// Accessing Array data using an Index
var arr = [1,2,3];
var data = arr[0]; // data = 1;

// Modify Array Data using an Index
var arr = [1,2,3];
arr[2] = 4; // now arr = [1,2,4]

// Nested Arrays
var arr1 = ["cars", 10];
var arr2 = ["bikes", 15];
var bigArr = [arr1, arr2]; // bigArr = [["cars", 10],["bikes", 15]]

// Accessing Nested Arrays using an Index
var arr = [[1,2,3],[4,5,6],[[7,8,9],10,11]];
arr[2]; // = [[7,8,9],10,11]
arr[2][0]; // = [7,8,9]
arr[2][0][1]; // = 8

/* ES6 Features */

// Reassign variables using Arrays;
let x = 7, y = 10;
[x,y] = [y,x];

// Slice Arrays using destructuring assignment
const source = [1,2,3,4,5,6,7];
const [a,b ...arr] = source;
console.log(arr); // it will print arr = [3,4,5,6,7]
```

Essential Methods

Strings

```

// Length of a String
var str = "example";
var length = str.length; // length = 4

//First character in a String
var str = "example";
var firstLetter = str[0]; // firstLetter = "e"

// Find nth character in a String
var str = "example";
var nChar = str[2]; // 3rd letter is "a"

// Last character in a String
var str = "example";
var lastLetter = str[str.length - 1]; // lastLetter = e

```

Arrays

```

// Add an element to the end of the array using push()
var arr = [1,2,3];
arr.push(4); // arr = [1,2,3,4]

var arr = [[1,2,3], [4,5,6]];
arr.push([7,8,9]); // arr = [[1,2,3], [4,5,6], [7,8,9]]

// Remove an element from the end of an array using pop()
var arr = [1,2,3];
arr.pop(); // arr = [1,2]

// Remove an element from the front of an array using shift()
var arr = [1,2,3];
arr.shift(); // arr = [2,3]

// Add an element to the front of an array using unshift()
var arr = [1,2,3];
arr.unshift(0); // arr = [0,1,2,3];

/* Advanced Methods */

// Remove elements using splice()
let arr = [1,2,3,4,5,6,7,8];
arr.splice(x,y); // x is the starting index, y is how many to delete from x
arr.splice(3,4); // remove 4 elements from index 3 (inclusive)
// arr = [1,2,3,8]

// Add elements using splice()
let arr = [1,2,3,4];
arr.splice(0,2,'y','z'); // all added at the same index , removed 2 elements
// arr = ['y','z',3,4];

// Copy elements using slice()
let arr1 = [1,2,3,4,5,6];
let arr2 = arr1.slice(x,y); // x is the starting index, y is final index(not inclusive)
let arr2 = arr1.slice(2,4);
// arr2 = [3,4];

// Copy an array using the spread operator

```

```

let arr1 = [1,2,3,4,5];
let arr2 = [...arr1]; // arr2 = [1,2,3,4,5]
arr1 = arr1.push(...arr2); // arr1 = [[1,2,3,4,5], [1,2,3,4,5]]
let arr3 = [0, ...arr2, 6, 7, 8]; // arr3 = [0,1,2,3,4,5,6,7,8]

// Check the index of an element in an array using indexOf()
let arr = [1,2,3,4,5];
arr.indexOf(4); // returns 3
arr.indexOf(7); // returns -1 as it does not exist in the array

```

Functions

```

// Example function with no parameters
function functionName(){
    console.log("Hello World");
}

// Example functions with parameters/arguments
function functionName(param1, param2){
    console.log(param1, param2);
}

// Example function with a return statement
function functionName(num){
    return num + 3;
}
var answer = functionName(5); // answer = 8
// a function without a return statement will give value of undefined

/* ES6 Feature */

// Utilise the destructuring assignment to pass object params into a function
const stats = {
    max: 56.78,
    median: 34.54,
    min: -0.75,
};

const half = ({max,min}) => (max + min)/2.0; // max and min taken directly out

```

Conditional Logic

```

// If statement Basic Structure
if (condition is true){ // does not need "= true"
    statement is executed;
}

// Using the Equality Operator
if (Variable == Value){
    statement is executed;
}

// Using the Strict Operator

```

```

3 === 3 // true
3 === '3' // false as the data types are different

// Using the Inequality Operator and Strict Inequality
1 != 2 // true
1 != '1' // false
1 !== '1' // true as the data types are different
1 != 1 // false

// Greater than/ Less than
5 > 3 // true
5 > '3' // true
5 >= 5 // true
5 >= '5' // true

5 < 3 // false
5 < '3' // false
5 <= 5 // true
5 <= '5' // true

// Using the And Operator &&
if (Variable == Value && Variable > Value){
    statement is executed;
}

// Using the Or Operator ||
if (Variable == Value || Variable > Value){
    statement is executed;
}

// Using Else If Statement
if (condition is true){ // does not need "= true"
    statement is executed;
}
else { // can add else if with more conditions
    statement is executed // iff condition is false
}

// Using Switch Case Statements
switch(String){
    case "x": execute statement; break; // break stops execution if case is satisfied
    case "y":
    case "z":
    case "a":
    case "b": execute statement; break; // matches all cases after last break to this oen
    default : defaultStatement; break; // always executed
}

// Returning boolean values
function isEqual(x,y){
    return a === b;
}

// Conditional Ternary Operator
// Structure: condition ? statement-if-true : statement-if-false ;
function findGreater(a,b){
    return a > b ? "a is greater" : "b is greater";
}

/* Use case to throw a specific error */
function isPositive(a) {

```



```

    if(a > 0){
        return 'YES';
    }
    else{
        throw (a === 0 ? new Error('Zero Error') : new Error('Negative Error'));
    }
}

```

Object Basics

```

// Object creation with a set of properties
var cat = {
    "name" : "cat",
    legs : 4, // can omit quotes for single word string word properties
    "tails" : 1,
    "enemies" : ["Water", "Dogs"],
    "favourite food" : "salmon"
};

// Access object properties with dot notation
var petName = cat.name; // petName = cat

// Access object properties with bracket notation
var petName = cat[name];
var favFood = cat["favourite food"];

// Access object properties with variables
function fav(str){
    var s = "favourite ";
    return s + str;
}
var prop = fav("food"); // prop now hold value for "favourite food"

var petName = "name"; // easy

// Updating object Properties
cat.name = "Fluffy"; // now "name" : "Fluffy" is the new property for the cat object
cat[name] = "Fluffy";

// Add new properties to the object
cat.noise = "meow meow";
cat[noise] = "meow meow";

// deleting properties from the object
delete cat.noise;

// Objects can be used as a key:value storage like a dictionary
var dict = { 1:"A", 2:"B", ..., 26: "Z"};

// Testing objects for properties
cat.hasOwnProperty("tails"); // true
cat.hasOwnProperty("toys"); // false

// Complex object structure
var fancyCat = [
    {
        "eyewear" : "monocle",

```

```

    "clothing" : ["socks", "shoes", "shirt", "tie"],
    "vehicle" : "cabriolet",
    "hungry" : false
  }
];

// Accessing nested objects
ourStorage.cabinet["top drawer"].folder2; // use a chain of dot and bracket notations
myPlants[1].list[1]; // accessing nested array objects

// Seeing if a Property exists in a given element of an array object
if(prop in arr[i]){
  execute statement;
}

/* ES6 Features */

// Define object literals using Object Property Shorthand
const createObject = (prop1, prop2) => {
  return({prop1, prop2});
};
// this method returns an object without the need for colon notation

//Write declarative functions
const car = {
  speed: 10,
  setSpeed(newSpeed) {
    this.speed = newSpeed;
  }
};

// Freezing an object so data cannot be changed
Object.freeze(obj);
obj.prop = value; // will be ignored. Mutation is not allowed
console.log(obj); // will work fine as it is not mutating the object

// We can utilise a destructuring assignment to extract values from Objects
const person = { name: 'J', age: 30 };

const name = person.name; // name = 'J'
const age = person.age; // age = 30

const { name, age } = person; // name = 'J', age = '30'
const { name : tag, age : timeAlive } = person; // tag = 'J', timeAlive = '30'

// Utilise destructuring assignment for nested variables
const user = {
  johnDoe : { age: 34, email: "jDoe@code.come" }
}

const {johnDoe : {age : userAge, email : userEmail}} = user;

```

Loops

```

// While Conditional Loop
var arr = [];
var i = 0

```

```

while (i < 5){
    arr.push(i);
    i++;
}

// Iterate using a For Loop
var arr = [];
for (var i = 0; i < 5; i++ /* can use i+=2 for more than 1*/){
    arr.push(i);
}

// Iterate using a Do ... While Loop
var arr = [];
var i = 0;
do {
    arr.push(i);
    i++;
} while (i < 5);

```

Recursion

```

// Replacing Loops using Recursion
function multiply(arr, n){
    var product = 1;
    for (var i = 0; i < n; i++) {
        product *= arr[i];
    }
    return product;
}

function multiply(arr, n){
    if(n <= 0){ // recursive functions use a base case to stop running
        return 1;
    }
    else {
        return multiply(arr, n-1) * arr[n - 1]; // often a decremental approach is applied
    }
}

```

ES6 Challenges

ECMAScript is a standardized version of JS which unifies the languages specs and features. All major browsers and JS-runtimes follow this spec. The term ECMAScript is interchangeable with the term JavaScript.

ECMAScript 5 (ES5) is a spec finalized in 2009. ECMAScript 6 (ES6) was released in 2015, and includes : Arrow Functions, Classes, Modules, Promises, Generators, Let and Const.

ES6 Supported on : Chrome, Firefox, Edge, Safari, Opera

ES7 Supported on : Chrome and Opera

Difference between var and let

```
// Declaring vars with the same name will overwrite the original var
var variable = 1;
var variable = 3;
console.log(variable); // it will print out 3 as the original was overwritten

// Using let will prevent overwrite and will throw an error
let variable = 1;
let variable = 3; // this will throw the error
```

Scope of let compared to var

let has a tighter scope than var, such as let will only return a value for the variable in the same block of code, such as in a for loop.

Const

```
// Const is a way to declare a variable that is read-only and cannot be reassigned.
const PET = "Dog"; // upper case names used for immutable variables
PET = "cat"; // this will return an error

// Const Arrays
const s = [2,5,7]; // no new array will be able to be called s
s[2] = 10; // this still works making s = [2,5,10]
```

Arrow Functions

```
// Arrow functions can be used to simply pass a function as an argument
const myFunc = function() {
  const myVar = "value";
  return myVar;
}

const myFunc = () => { // => is used to remove the need for function()
  const myVar = "value";
  return myVar;
}

// Arrow functions can also utilise parameters
const doubler = (item) => item * 2;

// Arrow functions can also have default values for parameters
const increment = (number, value = 1) => number + value; // default increment value is 1
```

Rest Parameter

```
// The Rest parameter allows functions to take a number of arguments
const sum = (x, y, z) => {
  const args = [x, y, z];
  return args.reduce((a, b) => a + b, 0);
}

const sum = (...args) => args.reduce((a, b) => a + b, 0); // (...args) is the rest

// the Spread operator allows expansion of an array or an expression
const arr1 = ['JAN', 'FEB', 'MAR', 'APR', 'MAY'];
let arr2;

arr2 = [...arr1]; // this copies the contents of arr1 into arr2
```

Template Literals

```
// Utilise template literals to create multi-line strings with interpolation
const item = {
  name: "Torch",
  batteryType: "AA"
};

const description = `This is a ${item.name},
It requires ${item.batteryType} batteries to function.`;
// note the use of `` (backtick) and the ${} to interpolate text
```

Class Syntax

```
// Utilise class syntax (not like java, Python etc) to define a constructor function
class Car{
  constructor(destination){
    this.destination = destination;
  }
}
const ford = new Car('London');
```

Getters and Setters

```
// Getters and Setters are used to obtain and change data and are part of the class
class DayOfYear{
  constructor(day){
    this._day = day; // _ is used to as syntax for a private variable
  }
  // getter
  get dayNum() {
    return this._day;
  }
}
```

```

    }
    // setter
    set dayNum(newDay) {
        this._day = newDay;
    }
}
const jan1 = new DayOfYear(1); // Setting in Fahrenheit scale
let temp = jan1.dayNum; // 1
jan1.dayNum = 26; // variable's value has been changed to 26 through set
temp = jan1.dayNum; // 26 has been returned through get

```

HTML with JS

```

<!-- Creating a Module Script which exports a JS file-->
<html>
  <body>
    <script type="module" src="index.js"></script> <!-- type="module" is necessary -->
  </body>
</html>

```

Export

```

// Use export to share code blocks
export const multiply =(x,y) =>{
    return x*y;
}

export {add, multiply, divide, subtract} // another method which groups exported code

// Export default is used as a fall back and when only one value is needed for export
export default function subtract(a,b) {
    return a - b;
}

```

Import

```

// Use import to utilise code blocks without writing them
import {multiply} from './operation_functions.js'; // placeholder filename
import subtract from './operation_functions.js'; // subtract is default

// Use * to import everything from a file
import * as operationModule from './operation_functions.js';

```

Promise

```
// Utilise a promise to do something asynchronously.
const makeRequest = new Promise ((resolve, reject) => {
  let response; // exaple of something to check
  if(condition){
    resolve("Promise was fulfilled"); // method checks outcome
  }
  else {
    reject("Promis was rejected"); // method checks outcome
  }
});

// Then is used to execute a function if the promise is resolved
makeRequest.then(result => {
  console.log(result);
});

// Catch is used to execute a function if the promise is rejected
makeRequest.catch(error => {
  console.log(error);
});
```

Regular Expressions in JavaScript

Regular expressions are special strings that represent a search pattern. Programmers utilise these expressions, known commonly as "regex", to handle text matching, searching and replacing. They are notorious for their poor legibility, but once we understand the special characters they become logical. These special characters are combined, through concatenation, to form a pattern, with which you will match other string to.

```
// Match to a single case
let myRegex = /Word/; // the criterion is "Word"
let testString = "RegexWordsAreCaseSensitive";
myRegex.test(testString); // .test() applies the regex and returns boolean

/regex/.test('string'); // concise syntax

// Match to multiple Cases
let myRegex = /Word|Regex|Are/; // using the | we can list the possibilities
let testString = "RegexWordsAreCaseSensitive";
myRegex.test(testString); // returns true

// Match to a case whilst ignoring letter cases
let myRegex = /Word|Regex|Are/i; // using i (insensitive flag) we ignore case
let testString = "RegexWordsAreCaseSensitive";
myRegex.test(testString); // returns true

// Use the .match() function to extract matches
let myRegex = /Word/; //
let testString = "RegexWordsAreCaseSensitive";
testString.match(myRegex); // returns ["Word"]
```

```

'string'.match(/regex/); // note it is the opposite structure to the .test() function

// Match to multiple occurrences of the same case
let myRegex = /Word/gi; // g (global flag) returns all matches
let testString = "Word, word, word";
testString.match(myRegex); // returns ["Word", "word", "word"]

// Match to anything with the wildcard period '.'
let myRegex = /wo./gi; // '.' can be put at the beginning to filter by final characters
let testString = "Word, world, worst, Winter";
testString.match(myRegex); // returns ["Word", "world", "worst"]

// Match specific characters
let myRegex = /[aei]/gi; // [] are used for the specific characters
let testString = "Ward, Wear, Worn, wire";
testString.match(myRegex); // returns ["Ward", "Wear", "wire"]

// Match an alphabetical sequence of characters
let myRegex = /[a-e]/gi; // - is used for the start and end (inclusive)
let testString = "Touring, tar, tuck, Time";
testString.match(myRegex); // returns ["tar", "Time"]

// Match a numerical sequence of integers
let myRegex = /[1-3a-c]/gi;
let testString = "Apple 1.561, Strawberry 2.798";
testString.match(myRegex); // returns ["Apple 1.561"]

// Match characters that are not specified
let myRegex = /^[^aeiou^0-5]/gi; // matches all non-vowels, including symbols like '!'
let testString = "a, b, c, @, e, 4, 5, 1";
testString.match(myRegex); // returns ["b", "c", "@"]

// Match a character occurring multiple times
let myRegex = /1+/gi;
let testString = "12, 115, 5151"; // note that in
testString.match(myRegex); // returns ["1", "11", "1", "1"]

// Match a character occurring zero or more times
let myRegex = /fun*/gi;
let testString = "fester, fun, full";
testString.match(myRegex); // returns ["f", "fun", "fu"]

// Match a character with Lazy Matching
let myRegex = /z[a-z]*?o/gi; // ? will return the smallest possible matching
let testString = "zoo, zoom, Zoom";
testString.match(myRegex); // returns ["zo", "zo", "Zo"]

// Match a pattern at the end of a string
let myRegex = /end$/; // $ is used to look at the end of strings
let testString1 = "This is the end";
let testString2 = "The end is not near";
myRegex.test(testString1); // returns true
myRegex.test(testString2); // returns false

// Match a pattern at the beginning of a string
let myRegex = /^First/i; // ^ is used to look at the start of strings
let testString1 = "Firstly, I would like to welcome you all";
let testString2 = "This isn't my first time";
myRegex.test(testString1); // returns true
myRegex.test(testString2); // returns false

```



```

// Match a pattern to all letters and numbers
let myRegex = /\w/; // "\w" is equivalent to [A-Za-z0-9_] and works efficiently
let testString1 = "12";
let testString2 = "letters";
myRegex.test(testString1); // returns true
myRegex.test(testString2); // returns true

// Match a pattern to everything other than letters and numbers
let myRegex = /\W/; // "\W" is equivalent to [^A-Za-z0-9_]
let testString1 = "12!";
let testString2 = "letters@";
testString1.match(myRegex); // returns ["!"]
testString2.match(myRegex); // returns ["@"]

// Match all numbers
let myRegex = /\d+/g; // "\d" is equivalent to [0-9]
let testString1 = "12 , twelve, 1";
testString1.match(myRegex); // returns ["12", "1"]

// Match all non-numbers
let myRegex = /\D+/g; // "\D" is equivalent to [^0-9]
let testString1 = "12 , twelve, 1";
testString1.match(myRegex); // returns ["twelve"]

// Match Whitespace
let myRegex = /\s+/g; // "\s" is equivalent to [\r\t\f\n\v]
let testString = "This sentence has weird spaces";
testString.match(myRegex); // returns [" ", " ", " ", " ", " "]

// Match Non-Whitespace
let myRegex = /\S+/g; // "\S" is equivalent to [^\r\t\f\n\v]
let testString = "This sentence has weird spaces";
testString.match(myRegex).length; // returns 26

// Match a pattern a specified amount of times
let myRegex = /bo{2,4}/g; // {x,y} x is the lower bound, y is the upper bound
// x or y can be left empty if just one bound is required
// {z} if z is on its own then the regex will match only for z occurrences
let testString = "bo, boo, boooo, booooo";
testString.match(myRegex); // returns ["boo", "boooo"]

// Match a pattern that may have some parts missing
let myRegex = /fury?/gi; // ? means that regex will match if there is no "y" present
let testString = "Fury, fur, Fuy";
testString.match(myRegex); // returns ["Fury", "fur"]

// Utilise Lookaheads to look for further matches on the same string
let myRegex1 = /t(?=o)/; // ?= is a positive lookahead
let myRegex2 = /t(?!o)/; // ?! is a negative lookahead
let testString1 = "to";
myRegex1.test(testString1); // returns true
myRegex2.test(testString1); // returns false

// Match a pattern to a mixed grouping of characters
let myRegex = /t(oot|un)ing/gi; // () encompass possibilities for the character
let testString = "Tuning, Tooting, touring";
myRegex.match(testString); // returns ["Tuning", "Tooting"]

// Match a Pattern using a capture group
let myRegex = /(\w+)\s\2/; // () encompasses the substring you look for
// \2 specifies the number of repeats

```

```

let testString = "12 12 12";
testString.match(myRegex); // returns ["12 12 12", "12"]

// Replace using capture groups
let myRegex = /not/;
let testString = "I work not hard";
testString.replace(myRegex, "very"); // returns "I work very hard"

/* Example 1, match a string where the character at the beginning
is the same as the character at the end */
// Assumptions: all strings are lowercase and are letters only, s >= 3 chars

let myRegex = /^[abcdefghijklmnopqrstuvwxyz]+\1$/;
// ^ first char matches
// () stores matching value
// [] matches to any value in brackets
// . matches any character
// + for more than 1 occurrence of the matched character
// \1 matches previously stored match
// $ ensures matched item is at the end of the string

```

Debugging

Testing is an essential aspect of programming. With Testing comes the need for Debugging which helps solve failed tests into passing tests. Debugging allows us to see exactly which part of our code is not functioning as it is supposed to.

Majority of coding errors look like:

1. Logical/Semantic Errors → **Hard to solve**
 - Mathematics was incorrect, code doesn't fulfil intended purpose
2. Runtime Errors
 - Out of Bounds exceptions, While loop that runs forever
3. Incorrect Syntax → **Easy To Solve**
 - Spelling mistakes, missing brackets and the elusive missed ";"

There are a number of tools that help programmers find bugs and solve them ideally. Debugging is from practice and the more you code the more practice at debugging you will get.

```

// Use console.log() to print any variable
var a = 4;
console.log(a); // returns 4 as it is printing before the increment takes place
a++;
console.log(a); // returns 5

// Use console.clear() to clear the console
console.clear();

```

```
// Use typeof to determine what data type a variable is
let a = 4
let b = "4";
let c = [4];
console.log(typeof a); // returns "number"
console.log(typeof b); // returns "string"
console.log(typeof c); // returns "object"
```

Eyes

These have been proven useful time and time again to detect humbling spelling errors as well as sketchy syntax. Make wise use of them and you will be gifted with working code and humility.

Be careful with using mixed " and ' quotations. Make sure that the ones you use to open are the same ones you use to close.

Data Structures

Data structures allow us to manage our data and how we access/manipulate it. They can vary in complexity and are built on one of 2 basic building blocks, arrays and dictionaries. Assume an Array is simply a list of items that are independant of each other with an index position. Adding arrays within arrays adds dimensions, a 2D array is an Array of Arrays, and hence has a more detailed index position. A Dictionary has a key-value structure where a Key is dependant on a Value. This allows a category based data structure where multiple objects can have the same category but different values for it., such as name, phone number and colour.

Arrays

```
// Example of a multi-dimensional array
let arr = [ //["name", isWorking?, age, 'hobby'] first level
  ["Stewie", false, 3, "World Domination"], // second level depth, array in an array
  ["Peter"], [true], [40], ["Beer"]], // third level depth,
  ["Morty", false, 14, "Magazines"],
  ["Rick", false, 65, "Space Travel"]
];
```

Dictionaries (built as arrays)

```
// A dictionary holds a key and a value, JS can have object properties that do similar
// You can emulate the sturcture of a table with multiple objects using them
let dict = { // note the use of {} for dictionaries as they are objects
  people : 4,
  realPeople : "0",
```

```

    diverse : true
  }

  console.log(dict.people); // returns 4

  // Addition of keys and values methods
  dict.smart = true; // adds a new key smart and sets its value to true

  let powers = 'hasPowers';
  dict[powers] = true; // adds a new key hasPowers and sets its value to true

  // Deleting a key and its value
  delete dict.powers; // removes the key and value

  // Check if an Object has a key
  dict.hasOwnProperty(realPeople); // returns true as the key is present
  'powers' in dict; // returns false as the key is not present

  // Iterate through objects
  for(let i in dict){ // i is an object in the dictionary or array
    //execute statement
  }

  // Create an array of keys only from an object
  Object.keys("input object");
  Object.keys(dict); // returns ["people","realPeople","diverse","smart"]

```

Object Oriented Programming

JavaScript has a unique ability to be utilised as an object oriented language as well as functional. Object Oriented programming is one of the most essential approaches in the software development process. Software development approaches dictate how a problem should be broken down and how its solution should be built up.

Object oriented code breaks down code into object definitions, known as classes, and their data and behaviour are monitored/manipulated using methods which are defined within or outside the class. Repeated code is reduced as objects can send/recieve data through these methods. Class structure allows new classes to inherit/send/recieve data from parent classes too.

This Section will Cover Object Oriented Programming Principles

Objects Advanced

visit objects basic for assumed knowledge

```

/* Create a method inside an object */
let shape = {
  name: 'Square',

```

```

    length: 5,
    eidth: 5,
    getArea: function() {return shape.length * shape.width;}
};

/* Utilise the "this" keyword to make reusable code */
let shape = {
    name: 'Square',
    length: 5,
    eidth: 5,
    getArea: function() {return this.length * this.width;}
};

/* Making Objects using a Monstructor */
function Rectangle(a,b) { //
    this.length = a; // note the change in syntax, we use = and ; instead of : and ,
    this.width = b;
    this.area = (this.length * this.width);
    this.perimeter = 2*(this.length + this.width);
}

let rec1 = new Rectangle(3,5); // rec 1 is a new rectangle object
rec1.area; // rec1's area is (3x5) = 15;

rec1 instanceof Rectangle; // returns true if rec1 is an instance of Rectangle
rec1.constructor === Rectangle; // returns true if rec1 is an instance of Rectangle

/* Utilise prototype to influence all instances' properties */
Rectangle.prototype.length = 5; // rec1.length = 5 now
Rectangle.prototype.color = "white"; // rec1.color = "white" -> a new property was added

// Separate an object's Own properties and Prototype Properties //
let ownProps = []; // own props are defined directly on the object
let prototypeProps = []; // prototype props are defined on the Constructor prototype
for(let prop in rec1){
    if(rec1.hasOwnProperty(prop)){
        ownProps.push(prop);
    }
    else {
        prototypeProps.push(prop);
    }
}

/* Set the Prototype to a new object which contains the existing properties */
Rectangle.prototype = {
    constructor: Rectangle, // define the constructor property always
    numSides: 4,
    is2D: true,
    grow: function() {
        this.length++;
        this.width++; // area and perimeter will change also, though not defined here
    }
};

Rectangle.prototype.isPrototypeOf(rec1); // returns true if rec1 inherits from Rectangle

// All objects in java have a prototype, this is also an object.
// A prototype can have its own prototype, forming a prototype chain
// this chain represents inheritance and supertypes
// Object is the supertype of both rectangle and rec1

```

```

/* Utilise Inheritance to produce DRY code */
// DRY = "Don't Repeat Yourself"

function Shape() { } // Supertype Shape gives grow to Square and Rectangle
Shape.prototype = {
  growLonger: function() {
    this.length++;
  }
};

function Square(sideLength){
  this.length = sideLength;
}
Square.prototype = {
  constructor: Square,
};

function Rectangle(length, width){
  this.length = length;
  this.width = width;
}
Rectangle.prototype = {
  constructor: Rectangle,
};

/* Create an object using the prototype of the supertype */
let square = Object.create(Shape.prototype);

/* Set a Child prototype to that of its parent supertype */
Square.prototype = Object.create(Shape.prototype); // inherits constructor property
Square.prototype.constructor = Square; // ensures constructor is Square

/* Add a function unique to the child */
Square.prototype.area = function() {
  this.area = (this.length*this.length);
}

/* Add an overriding function to the child */
Square.prototype.growLonger = function() { // overrides the supertype method : growLonger
  this.length = this.length + 2;
}

/* Use Mixins to when inheritance is not ideal for vastly different objects */
let cryMixin = function(obj) {
  obj.cry = function() {
    console.log(":", "(");
  }
};

let person = {
  name: "Rick",
  numLegs: 2
};

let onion = {
  private type: "Shalot", // this makes the vairbale private and protected
  colour: "white"
};

cryMixin(onion);
cryMixin(plane); // this attaches the function to both onion and plane

```

```

/* Classes with inheritance */
class Rectangle { // base class of rectangle
  constructor(w, h) {
    this.w = w;
    this.h = h;
  }
}
Rectangle.prototype.area = function() { // adds property with function to prototype
  return(this.w*this.h);
};

class Square extends Rectangle { // "exetends" makes Rectangle a super class of Square
  constructor(l) {
    super(l,l); // passes l variable into the super class constructor
    this.h = l;
    this.w = l;
  }
}

// Class definition with a function with it
class Polygon{
  constructor(arr){ // array of side lengths passed into constructor
    this.sides = arr;
  }
  perimeter() {
    return this.sides.reduce((total, side) => total + side);
  }
}

/* An IIFE is an Immediately Invoked Function Expression */
// These are functions that are executed right away
(function(){ // unique syntax at the beginning and end of the function
  console.log("I am an IIFE, I am cool");
})(); // these parenthesis cause the function to be invoked immediately

/* IIFEs are used to create Modules at the begining */
// Modules contain related functionality as a single object
let motionModule = (function () {
  return{
    driveMixin: function(obj){
      obj.drive = function() {
        console.log("Driving on the road");
      };
    },
    cycleMixin: function(obj){
      obj.cycle = function() {
        console.log("Cycling on the road");
      };
    }
  }
})();

```

Functional Programming

Functional programming is a different approach to software development based on evaluation of functions by mapping input to output to produce a result. Combining basic functions results in increasingly complex software structure.

This section will cover the basics of functional programming and show you the true benefits of utilising functional programming to build clean, concise code.

Core Principles:

- Functions are independent from the state of the program or global variables. They only depend on the arguments passed into them in order 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

$$INPUT \rightarrow PROCESS \rightarrow OUTPUT$$

Functional Programming Terminology

Functional Programming includes:

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, mutations, to the state of the program outside the function are carefully controlled

```
/* Scenario: Ordering chips for friends */
// functions that return a string representing a portion of a type of chips
const orderSaltyChips = () => "saltyChips"; // no need for "return" or "{}"
const orderSpicyChips = () => "spicyChips";
const orderCheesyChips = () => "cheesyChips";

// Function that takes a number of orders and the type of chips to return an array
const getChips = (orderChips, numOfPortions) =>{ // orderChips is a function argument
  let chipOrder = []; // declare an array to put results in
  for(let portions = 0; portions< numOfPortions; portions++){
    let chips = orderChips();
    chipOrder.push(chips);
  }
  return chipOrder;
}
```

Callback: In the scenario above we have something called a *Callback*. These are functions that are passed into another function as a parameter which determine the

invocation of that function.

First Class Functions: In JavaScript all functions are *first class* functions which work.

Higher Order: Functions that take other functions as their arguments or returns a function as a return value are called higher order functions.

Imperative Programming: This approach to coding gives the computer a set of statements to execute in order to perform a given task. They often change the state of the program, changing global variables etc or elements in an array.

```
/* Example of Imperative Programming */
// Program to add 2 numbers together
let num1 = 10;
let num2 = 5;

console.log(num1 + " + " + num2 + " = " + (num1+num2)); // prints " 10 + 5 = 15"
```

Declarative Programming: This approach gives the computer a function to carry out which will perform the task you want. IE JS provides (map) which iterates over an array like a for loop but without the chance of a semantic error. Calling these functions will lead to changes in the state of the program.

```
/* Example of Declarative Programming */
const add = (num1, num2) => {return (num1+num2)};
```

Mutations: A key principle of functional programming is not to change things. Manipulating global variables too much can lead to bugs, preventing anything from changing alleviates that risk. these manipulations are mutations.

Side Effects: These are effects that are caused by mutations, like bugs, runtime errors etc

Pure Functions: Functions that do not contain mutations and do not cause side effects. These functions clearly declare dependencies.

```
/* Example of a Pure Function */
let fixedValue = 10; // Global value that we wont change
function decrement = (value) => { // declaration of argument means dependency is realised
  return value - 1; // does not change the input
}
```

Arity: The arity of a function is the number of arguments it requires

Currying: This is a process where a function of N arity is converted to N functions of arity 1

```
/* Example of Currying a Function*/
// Uncurried function of arity 2
function add(x,y){
    return x+y;
}
// Curried function
function addCurry(x){
    return function(y){
        return x+y;
    }
}

// Curried function in ES6
const addCurry = x => y=> x+y;
addCurry(x)(y); // returns x+y
var funcForY = addCurry(1); // saves a function call into a variable
console.log(funcForY(3)); // returns 4(3+1), useful if all arguments are not at hand
```

Functional Manipulation of Arrays

Map

```
/* Extract data from an Array using the Map method */
let arr = [
    {num: 1, chr: 'a', color:"orange"},
    {num: 2, chr: 'b', color:"black"},
    {num: 3, chr: 'c', color:"white"},
    {num: 4, chr: 'd', color:"red"},
    {num: 5, chr: 'e', color:"blue"}
];
const nums = arr.map(num => arr.num); // nums = [1,2,3,4,5]
const chars = arr.map(char => arr.chr); // chars = ['a','b','c','d','e']
const arr2 = arr.map(({num: n, chr: c})=>({n,c}));
/* arr 2 = [
    {num: 1, chr: 'a'},
    {num: 2, chr: 'b'},
    {num: 3, chr: 'c'},
    {num: 4, chr: 'd'},
    {num: 5, chr: 'e'}
]; */
const arr3 = arr.map(entry => {return {
    num: entry.num,
    chr: entry.chr};
}); // another method to make arr2
```

Filter

```

/* Extract data from an Array using the filter method */
let arr = [
  {num: 1, chr: 'a', color:"orange"},
  {num: 2, chr: 'b', color:"black"},
  {num: 3, chr: 'c', color:"white"},
  {num: 4, chr: 'd', color:"red"},
  {num: 5, chr: 'e', color:"blue"}
];
const cUnder4 = arr.filter(c => (c.num < 4)); // cUnder4 includes elements where num<4
const cOver2 = arr.filter(entry => {return entry.num>2;});
// cOver2 includes elements where num>2

```

Concat

```

/* The concat method will join two arrays */
let arr1 = [1,2,3];
let arr2 = [4,5,6];
let arr3 = arr1.concat(arr2); // arr3 = [1,2,3,4,5,6];

```

Reduce

```

/* The reduce method processes an array and solves a given problem */
const arr1 = [
  {id: 'A', num: 12},
  {id: 'B', num: 10},
  {id: 'C', num: 15},
  {id: 'D', num: 18},
  {id: 'E', num: 25},
  {id: 'F', num: 30}
];

const sumNums = arr1.reduce((sum,i) => sum + i.num,0); // 110
const objDict = arr1.reduce((obj, i) => {
  obj[i.id] = i.num;
}, {}); // returns a dictionary with id as keys and nums as values
// returns {A:12, B:10, C:15, D:18, E:25, F:30}

```

Sort

```

/* the sort method allows us to change the order of elements in an array */
function ascendOrd(arr){
  return arr.sort(function(x, y) {
    return x - y;
  });
}
ascendOrd([2, 4, 1, 3, 5]); // returns [1,2,3,4,5]

// example : order an array of characters in alphabetical order
function alphaOrd(arr) {
  let res = [...arr]; // res is a copy of the array, in order to not mutate the original

```

```

    return arr.sort(function(x,y) {
        return x === y ? 0 : x > y ? 1: -1
    });
}
alphaOrd(["c", "a", "l", "z", "y", "x"]); // returns ["a","c","l","x","y","z"]

```

Split & Join

```

/* The split method separates words in a string into an array to parse through */
let str = "This is-a,Weird String";
let strArr = str.split(/\W/); // strArr = ["This","is","a","Weird","String"];

/* The join method combines an array of String words into a single string */
let strFix = strArr.join(" "); // strFix = "This is a Weird String"

// notice the use of " " and Regex (/W/), both can be applied as criteria for join/split

```

Every

```

/* The every method checks if all elements satisfy a test */
let arr1 = [1,2,3,-4];
const checkPos = (arr) => {
    return arr.every(function(val) {
        return val>0;
    });
}
console.log(checkPos(arr1)); // returns false as -4 is in the array and is not positive

```

Some

```

/* The some methods checks if any element statisfies a test */
let arr1 = [-1,-2,-3,4];
const checkIfOneIsPos = (arr) => {
    return arr.some(function(val) {
        return val>0;
    });
}
console.log(checkIfOneIsPos(arr1)); // returns true as 4 is positive

```