Basic JavaScript, DOM

Basic JavaScript

comments: // or /\* stuff \*/

data types:

1. undefined

2. null

3. boolean

4. string

5. symbol

6. number

7. object.

“use strict” to catch coding mistakes and unsafe actions.

Declaring variable

**var keyword**

var ourName; // have variable *undefined* if not given initial variable.

var ourName = “Dog”;

Especially problematic if you declare it in a for loop to count:

for (var ii = 0; ii < 3; ii++){

// global variable ii is now incremented.

}

**let keyword**

only defined in scope nearest enclosing block. Global if outside any block.

block scoping

let ourName = “scopedDog”;

**const keyword**

defined once, never redefined. Use capital letters.

const OURNAME = “ScopedDog”;

if assign OURNAME = “Fish” => TypeError!

Const arrays are still mutable, but no reassignment allowed!

"use strict";  
const s = [5, 6, 7];  
s = [1, 2, 3]; // throws error, trying to assign a const  
s[2] = 45; // works just as it would with an array declared with var or let

But if want total immutability, use Object.freeze(obj);

**Temporal Dead Zone:**

for **let** and **const**, both are hoisted but but cannot be accessed until after it has been declared. You will have ReferenceError if you call it before declared. While with **var**, you can call it before declared and get undefined.

Variables and functions are case sensitive and variables are typically in camelCase.

Numbers

*is a wrapper object so that you can work with numerical values*.

operations are same as other languages.

I++ = i+1;

numbers can be decimal points, no need to do anything different.

% remainder operator is not modulus, it does not work with negative numbers.

MyVar **+=** is myVar = myVar + num;

myVar **-=** is myVar = myVar – num;

myVar **\*=** is myVar = myVar \* num;

myVar **/=** is myVar/num;

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

Not all real numbers can accurately be represented in floating point. This can lead to rounding errors. **Loss of Significance Errors**

Strings

*global object.*

var stringName = “Strings can be added together.”;

if you want to use the quotations marks “ or ‘ inside your string, you need to use the **escape** from the quotes using an escape sequence”.

**Escape Sequences**

| Code | Output |
| --- | --- |
| \' | single quote |
| \" | double quote |
| \\ | backslash |
| \n | newline |
| \r | carriage return |
| \t | tab |
| \b | backspace |
| \f | form feed |

**Length of String**

.length will get the length of string literal. Last character of string is

stringName[stringName.length-1];

**Indexing a String**

Indexing of string is through [x]; // starts from 0

**Immutability of String**

strings are *immutable* meaning that once defined, they cannot be changed.

E.g. var myStr = “Bob”;

myStr[0] = “J”; =>NO CHANGE OCCURS!

Need to reassign to get change…

myStr = “Job”;

**Copy A String**

String.slice();

Good way to copy a string, as it returns a new copied version of the string so that you can modify the values leaving the old string unaffected. This is done instead because when you assign a variable to a string, all you do is set a reference to the original, and your copy is immutable.

**Finding Occurrences of Words in Strings**

Regular expressions are used to find certain words or patterns inside of strings.

For example, if we wanted to find the word the in the string The dog chased the cat, we could use the following regular expression: /the/gi

Let's break this down a bit:

/ is the start of the regular expression.

the is the pattern we want to match.

/ is the end of the regular expression.unctions.

g means global, which causes the pattern to return all matches in the string, not just the first one.

i means that we want to ignore the case (uppercase or lowercase) when searching for the pattern.

andCount = testString.match(expression).length;

for **numbers** use expression: var expression = /\d+/g; // the + symbol means look for more than one.

For **whitespace** use expression: var expression = /\s+/g;

For any **non-whitespace** use expression: var expression = /\S/g;

Conditional Statements

**if statements**

if (condition){

statements;

}

wasThatTrue and isItTrue are boolean conditions uses in if statements.

**== Equality operator**

JavaScript can compare over two different data types such as 1 == ‘1’ = TRUE.

**=== Strict Equality Operator**

JavaScript has an operator that compares data types as well. Here 1 === ‘1’ = FALSE.

*!= inequality operator.*

*!== strict inequality operator.*

&& AND

|| OR

remember to string your if and else if statements from smallest to biggest to capture the best resolution!

**Switch statements**

switch (num){

case value1:

statement1;

break;

case value2:

statement2;

break;

…

case valueN:

statementN;

break;

default:

statement;

}

a cool trick using switch statements:

switch(val) {  
case 1:  
case 2:  
case 3:  
result = "1, 2, or 3";  
break;  
case 4:  
result = "4 alone";  
}

refactoring trick!

don’t need if or else statements if return [equality statement] does the same.

Loops

**for loops**

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

statements;

}

**while loops**

while(condition){

statements;

}

**do...while loops**

do{

// statements

} while(condition);

will run through all the statements in the do loop before checking the while condition, and if true, will repeat do statements until while condition is false.

Functions and Scoping

**Scope**

local vs global:

global variables are just variables defined outside all functions and blocks.

Local variables are just within a block. Local variables with the same name as global variables will override the global variable.

**functions (nonES6)**

function functionName(input variables){

return variable;

}

// if no return, then undefined is returned.

**arguments object**

you can get the arguments input into a function by going arguments[x] within the function.

**Return Early Pattern (for debugging)**

function myFun() {  
  console.log("Hello");  
  return "World";  
  console.log("byebye")  
}  
myFun();

So that if there is an error in the function, and function does not return, you will have a console output.

Objects

similar to arrays but instead of using indexes to access and modify data, you access data through *properties*. Nearly everything in Javascript is an object, except for the primitives (String, Number, and Boolean) which are immutable, but even these have wrappers to make them behave like objects.

*object layout*

var objectName ={

“prop1”: “stuff”,

“prop2”: 2,

“prop3”: 1,

“prop4”: [“Water”, “Dogs”]

};

accessing properties: *objectName.prop1*;

or *objectName[“prop1”]*;

Here is an example of using a variable to access a property:

var someProp = "propName";  
var myObj = {  
propName: "Some Value"  
}  
myObj[someProp]; // "Some Value"

**overwrite properties**

by going *objectName.prop1 = “NEW THING”;*

**add properties**

by going *objectName.newprop = “NEW Thing!”*;

**delete properties**

by going *delete objectName.prop1*;

**check if property exist**

by.hasOwnProperty() returns true or false.

You can also **nest objects** together

**creating objects using constructor functions**

var Car = function() {  
  this.wheels = 4;  
  this.engines = 1;  
  this.seats = 5;  
};

As functions are actually just objects.

**making instances of objects with constructor function**

var myCar = new Car();

another cool example:

// Setup

function phoneticLookup(val) {

var result = "";

var lookup ={

alpha: "Adams",

bravo: "Boston",

charlie: "Chicago",

delta: "Denver",

echo: "Easy",

foxtrot: "Frank"

}

result = lookup[val];

return result;

}

phoneticLookup("charlie");

**map method**  is used to iterate through an array using multiple processes simultaneously.

OldArray.map(function(val)){ //multiplies all values in array by 4

return val\*4;

}

**reduce method** is used to operate cumulatively on each value of array.

var singleVal = array.reduce(function(previousVal, currentVal) {  
  return previousVal - currentVal;  
}, 0);

**filter method** is used to filter out unwanted variables.

array = array.filter(function(val) {  
  return val !== 5;  
});

**sort method** is used to sort array, it changes the original array and typically you should do a compare function inside of it too so that it knows how you want to sort it.

var array = [1, 12, 21, 2];  
array.sort(function(a, b) {  
  return a - b;  
});

**reverse method** is a method that just reverses an array, it alters the original array.

var myArray = [1, 2, 3];  
myArray.reverse();

**concat method** is a method that joins an array to the end of the first array.

newArray = oldArray.concat(otherArray);

**split method** is a method to split strings into many strings and put them into an array.

Var array=string.split(‘s’);

**join method** is a method that takes an array of strings and then joins them together to form one string. You can place an argument between each string.

Var salad = veggies.join(“ and “);

**JSON**

[JavaScript Object Notation](http://www.json.org/) or JSON is a related data interchange format used to store data.

{  
"artist": "Daft Punk",  
"title": "Homework",  
"release\_year": 1997,  
"formats": [   
"CD",  
"Cassette",  
"LP"  
],  
"gold": true  
}

**Unique Objects**

var Car = function(wheels, seats, engines) {  
  this.wheels = wheels;  
  this.seats = seats;  
  this.engines = engines;  
};

Objects can have their ownfunctions called **methods**.

var Car = function() {

// this is a **private** variables  
 var speed = 10;

// these are **public** methods  
 this.accelerate = function(change) {  
 speed += change;  
 };

this.decelerate = function() {  
 speed -= 5;  
 };

this.getSpeed = function() {  
 return speed;  
 };  
};

Math Object

**Random Numbers**

between 0 to 1 *Math.random()*

**floor fractions**

*Math.floor()*

console.log("Hi World"); // prints to console

**Random Numbers Within a Range**

Math.floor(Math.random() \* (max - min + 1)) + min

**parseInt()**converts numbers in string to integers. Returns NaN if first character of string cannot be converted to number.

parseInt(str, radix); where radix converts it into radix number system. e.g. 2 = binary.

**Ternary operator**condition ? Statement-if-true: statement-if-false;

Can be used to chain together multiple conditions e.g.

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

ES6

**Arrow Functions**

for unnamed functions, especially passing function as argument to another function. e.g.

const myFunc = function() {  
  const myVar = "value";  
  return myVar;  
}

Arrow functions are a new way of writing anonymous functions.

const myFunc = () => {  
  const myVar = "value";  
  return myVar;  
}

**Pass arguments into Arrow Functions**

const doubler = (item) => item \* 2;

can have more than one argument as well.

**Default Parameters for Functions**

(defaultParameterName = “DefaultParameter”)=>{}

Will work for all functions. New ES6 rule.

**Rest Operator …args**

function functName(...args){

// the ...args takes in an indefinite number of arguments, and puts it into the args array.

}

**Spread Operator ...args**

Expands array into its parts. e.g. [a,b,c] becomes a, b, c.   
Useful for expanding an array to find max e.g.

var maximus = Math.max(...arr); // because Math.max only accepts comma separated values, not arrays Math.max(a,b,c);

// in ES5, used apply() function to fix this, but spread operator much neater. Cannot be used on its outside a function! Dog = [a,b,c]; cat = ...Dog; // SYNTAX ERROR!

**Destructuring Assignment**

breaks out the properties of an object or the contents of an array and puts them into their own variables e.g.

var voxel = {x: 3.6, y: 7.4, z: 6.54 };

const { x, y, z } = voxel; // x = 3.6, y = 7.4, z = 6.54, can use x instead of voxel.x;

**Destructuring Assignments to assign variables from Nested Objects**

const a = {  
  start: { x: 5, y: 6},  
  end: { x: 6, y: -9 }  
};  
const { start : { x: startX, y: startY }} = a;

can do it for nested objects, start is taken out of a and then placed into {x:startX, y:startY} which is sort of like:

{x:startX, y:startY} = a.start;

So startX == 5 and startY == 6.

**Destructuring Assignment to Assign Variables from Arrays**

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

We can also access the value at any index in an array with destructuring by using commas to reach the desired index:

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

This is better than the spread operator in some cases because you can choose what you want unpacked.

**Using the Destructuring Assignment with Rest to Slice for Last Elements**

const [a, b, ...arr] = [1, 2, 3, 4, 5, 7];  
console.log(a, b); // 1, 2  
console.log(arr); // [3, 4, 5, 7]

**Destructuring Assignment to Pass Object as Parameters**

const profileUpdate = (profileData) => {  
  const { name, age, nationality, location } = profileData;  
  // do something with these variables  
}

Refactoring…

const profileUpdate = ({ name, age, nationality, location }) => {  
  /\* do something with these fields \*/  
}

Why not just pass it as parameters in a function? Because Destructuring creates a new copy of the data being passed in.

**Template Literal**

special type of string that makes creating complex strings easier.

Sort of like how strings are done in C where you can add variables into the string.

const person = {  
  name: "Zodiac Hasbro",  
  age: 56  
};  
  
// Template literal with multi-line and string interpolation  
const greeting = `Hello, my name is ${person.name}!  
I am ${person.age} years old.`;  
  
console.log(greeting); // prints  
// Hello, my name is Zodiac Hasbro!  
// I am 56 years old.

Firstly, the example uses backticks (`), not quotes (' or "), to wrap the string.

Secondly, notice that the string is multi-line, both in the code and the output. This saves inserting within strings.

The ${variable} syntax used above is a placeholder. Basically, you won't have to use concatenation with the + operator anymore. To add variables to strings, you just drop the variable in a template string and wrap it with ${ and }. Similarly, you can include other expressions in your string literal, for example ${a + b}.

**Object Literals Declaration Using Simple Fields**

First of all, Object Literals are sort of like objects but are not.

Consider the following code:

const getMousePosition = (x, y) => ({  
  x: x,  
  y: y  
});

getMousePosition is a simple function that returns an object containing two fields.

ES6 provides the syntactic sugar to eliminate the redundancy of having to write x: x. You can simply write x once, and it will be converted tox: x (or something equivalent) under the hood.

Here is the same function from above rewritten to use this new syntax:

const getMousePosition = (x, y) => ({ x, y });

**Functions Within Objects**

also called Concise Declarative Functions.

Previously with ES5:

const person = {  
  name: "Taylor",  
  sayHello: function() {  
    return `Hello! My name is ${this.name}.`;  
  }  
};

With ES6, You can remove the function keyword and colon altogether when defining functions in objects. Here's an example of this syntax:

const person = {  
  name: "Taylor",  
  sayHello() {  
    return `Hello! My name is ${this.name}.`;  
  }  
};

**Defining Objects from Function using Class Syntax**

In ES5, we usually define a constructor function, and use the new keyword to instantiate an object.

var SpaceShuttle = function(targetPlanet){  
  this.targetPlanet = targetPlanet;  
}  
var zeus = new SpaceShuttle('Jupiter');

The class syntax simply replaces the constructor function creation:

class SpaceShuttle {  
  constructor(targetPlanet){  
    this.targetPlanet = targetPlanet;  
  }  
}  
const zeus = new SpaceShuttle('Jupiter');

**Getters and Setters to Control Access to Object**class Book {  
  constructor(author) {  
    this.\_author = author;  
  }  
  // getter  
  get writer(){  
    return this.\_author;  
  }  
  // setter  
  set writer(updatedAuthor){  
    this.\_author = updatedAuthor;  
  }  
}  
const lol = new Book('anonymous');  
console.log(lol.writer);  // anonymous  
lol.writer = 'wut';  
console.log(lol.writer);  // wut

Notice the syntax we are using to invoke the getter and setter - as if they are not even functions.

Getters and setters are important, because they hide internal implementation details.

**Import vs Require**

In the past, the function require() would be used to import the functions and code in external files and modules. While handy, this presents a problem: some files and modules are rather large, and you may only need certain code from those external resources.

ES6 gives us a very handy tool known as import. With it, we can choose which parts of a module or file to load into a given file, saving time and memory.

import { countItems } from "math\_array\_functions"

A description of the above code:

import { function } from "file\_path\_goes\_here"  
// We can also import variables the same way!

There are a few ways to write an import statement, but the above is a very common use-case.

Note  
The whitespace surrounding the function inside the curly braces is a best practice - it makes it easier to read the import statement.

Note  
The lessons in this section handle non-browser features. import, and the statements we introduce in the rest of these lessons, won't work on a browser directly. However, we can use various tools to create code out of this to make it work in browser.

Note  
In most cases, the file path requires a ./ before it; otherwise, node will look in the node\_modules directory first trying to load it as a dependency.

**Export to Reuse Code Block**

In the previous challenge, you learned about import and how it can be leveraged to import small amounts of code from large files. In order for this to work, though, we must utilize one of the statements that goes with import, known as export. When we want some code - a function, or a variable - to be usable in another file, we must export it in order to import it into another file. Like import, export is a non-browser feature.

The following is what we refer to as a named export. With this, we can import any code we export into another file with the import syntax you learned in the last lesson. Here's an example:

const capitalizeString = (string) => {  
  return string.charAt(0).toUpperCase() + string.slice(1);  
}  
export { capitalizeString } //How to export functions.  
export const foo = "bar"; //How to export variables.

Alternatively, if you would like to compact all your export statements into one line, you can take this approach:

const capitalizeString = (string) => {  
  return string.charAt(0).toUpperCase() + string.slice(1);  
}  
const foo = "bar";  
export { capitalizeString, foo }

Either approach is perfectly acceptable.

**Use \* to Import Everything from a File**

Suppose you have a file that you wish to import all of its contents into the current file. This can be done with the import \* syntax.

Here's an example where the contents of a file named "math\_functions" are imported into a file in the same directory:

import \* as myMathModule from "math\_functions";  
myMathModule.add(2,3);  
myMathModule.subtract(5,3);

And breaking down that code:

import \* as object\_with\_name\_of\_your\_choice from "file\_path\_goes\_here"  
object\_with\_name\_of\_your\_choice.imported\_function

You may use any name following the import \* as portion of the statement. In order to utilize this method, it requires an object that receives the imported values. From here, you will use the dot notation to call your imported values.

**Export Default**

There is another export syntax you need to know, known as export default. Usually you will use this syntax if only one value is being exported from a file. It is also used to create a fallback value for a file or module.

Here is a quick example of export default:

export default function add(x,y) {  
  return x + y;  
}

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

**Import a Default Export**

In the following example, we have a function, add, that is the default export of a file, "math\_functions". Here is how to import it:

import add from "math\_functions";  
add(5,4); //Will return 9

The syntax differs in one key place - the imported value, add, is not surrounded by curly braces, {}. Unlike exported values, the primary method of importing a default export is to simply write the value's name after import.

Regular Expressions

If you want to find the word "the" in the string "The dog chased the cat", you could use the following regular expression: /the/. Notice that quote marks are not required within the regular expression.

JavaScript has multiple ways to use regexes. One way to test a regex is using the .test() method. The .test() method takes the regex, applies it to a string (which is placed inside the parentheses), and returns true or false if your pattern finds something or not.

let testStr = "freeCodeCamp";  
let testRegex = /Code/;  
testRegex.test(testStr);  
// Returns true

Will only match exactly to the Regex. Different capitalisation will be false.

**Different Possibilities:**

This is powerful to search single strings, but it's limited to only one pattern. You can search for multiple patterns using the alternation or OR operator: |.

This operator matches patterns either before or after it. For example, if you wanted to match "yes" or "no", the regex you want is /yes|no/.

**Ignore Case While Matching**

You can match both cases using what is called a flag. There are other flags but here you'll focus on the flag that ignores case - the i flag. You can use it by appending it to the regex. An example of using this flag is /ignorecase/i. This regex can match the strings "ignorecase", "igNoreCase", and "IgnoreCase".

**Extracting Your Matches**

So far, you have only been checking if a pattern exists or not within a string. You can also extract the actual matches you found with the .match() method.

To use the .match() method, apply the method on a string and pass in the regex inside the parentheses. Here's an example:

"Hello, World!".match(/Hello/);  
// Returns ["Hello"]  
let ourStr = "Regular expressions";  
let ourRegex = /expressions/;  
ourStr.match(ourRegex);  
// Returns ["expressions"]

**Find More than First Match**

To search or extract a pattern more than once, you can use the g flag.

let repeatRegex = /Repeat/g;  
testStr.match(repeatRegex);  
// Returns ["Repeat", "Repeat", "Repeat"]

If want more than one flag, then just go /gi.

**Matching with WildCard Period .**

Use the **full stop character.** to match any character.

For example, if you wanted to match "hug", "huh", "hut", and "hum", you can use the regex /hu./ to match all four words.

**Matching Single Character with Multiple Possibilities**

Use **Character Sets** that can define a group of characters you wish to match inside square ([ and ]) brackets.

For example, you want to match "bag", "big", and "bug" but not "bog". You can create the regex /b[aiu]g/ to do this. The [aiu] is the character class that will only match the characters "a", "i", or "u".

**Matching Range of Characters**

[a-z]

**Negated Character Sets**

To create a negated character set, you place a caret character (^) after the opening bracket and before the characters you do not want to match.

For example, /[^aeiou]/gi matches all characters that are not a vowel. Note that characters like ., !, [, @, / and white space are matched - the negated vowel character set only excludes the vowel characters.

**Match Repeating Characters**

Add + after the repeating character that you want to match.

For example, /[s+]/g matches “Mississippi” returns [“ss”,”ss”];

**Match Characters that Appear Zero or More Times**

There's also an option that matches characters that occur zero or more times.

The character to do this is the asterisk or star: \*.

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

**Lazy Matching**

Use the ? Character to return the lazy matching of the character (the smallest substring fit for the pattern).

**Finding Matches at the Beginning of the String**

/^beginning/

**Finding Matches at the End of the String**

/end$/

**Matching all Letters and Numbers**

\w is use to match all letters and numbers, and character ‘\_’.

**Matching Everything But Letters, Numbers and character “\_”**

\W

**Matching All Numbers**

\d

**Matching All Non-Numbers**

\D

**Matching Whitespace**

\s similar to the character class [^ \r\t\f\n\v]

**Matching Non-Whitespace**

\S

**Specify Upper and Lower Numbers of Matches**

You can specify the lower and upper number of patterns with quantity specifiers. Quantity specifiers are used with curly brackets ({ and }). You put two numbers between the curly brackets - for the lower and upper number of patterns.

For example, to match only the letter a appearing between 3 and 5 times in the string "ah", your regex would be /a{3,5}h/.

**Exact Matches**

{x\_times\_min, x\_times\_max}

or {x\_times}

**Check for All or None**

Sometimes the patterns you want to search for may have parts of it that may or may not exist. However, it may be important to check for them nonetheless.

You can specify the possible existence of an element with a question mark, ?. This checks for zero or one of the preceding element. You can think of this symbol as saying the previous element is optional.

For example, there are slight differences in American and British English and you can use the question mark to match both spellings.

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

**Positive and Negative Lookaheads**

Lookaheads are patterns that tell JavaScript to look-ahead in your string to check for patterns further along. This can be useful when you want to search for multiple patterns over the same string.

There are two kinds of lookaheads: positive lookahead and negative lookahead.

A positive lookahead will look to make sure the element in the search pattern is there, but won't actually match it. A positive lookahead is used as (?=...) where the ... is the required part that is not matched.

On the other hand, a negative lookahead will look to make sure the element in the search pattern is not there. A negative lookahead is used as (?!...) where the ... is the pattern that you do not want to be there. The rest of the pattern is returned if the negative lookahead part is not present.

Lookaheads are a bit confusing but some examples will help.

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

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

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

**Capture Groups**

Some patterns you search for will occur multiple times in a string. It is wasteful to manually repeat that regex. There is a better way to specify when you have multiple repeat substrings in your string.

You can search for repeat substrings using capture groups. Parentheses, ( and ), are used to find repeat substrings. You put the regex of the pattern that will repeat in between the parentheses.

To specify where that repeat string will appear, you use a backslash (\) and then a number. This number starts at 1 and increases with each additional capture group you use. An example would be \1 to match the first group.

The example below matches any word that occurs twice separated by a space:

let repeatStr = "regex regex";  
let repeatRegex = /(\w+)\s\1/;  
repeatRegex.test(repeatStr); // Returns true  
repeatStr.match(repeatRegex); // Returns ["regex regex", "regex"]

Using the .match() method on a string will return an array with the string it matches, along with its capture group.

Debugging

Console.log();

**typeof variable** returns type of variable.

Look for mispelled variables.

Ensure that parentheses, brackets, braces, and quotes are closed.

Ensure that single and double quotation marks are not mixed together. Use escape character \ if necessary.

Ensure that equality operator is used instead of assignment operator.

Ensure that functions are called with parentheses.

Ensure that arguments are entered into functions in the right order.

**Off By One Errors (OBOE)** forgetting that Javascript uses 0th indexing, so set your functions to go from 0 to less than length if you want to access array.

Forgetting to reinitialise variable when setting values into array.

Infinite looping because your loop breaking statement does not have a valid condition to break out of.

Data Structures

Simplest data structure:

**one-dimensional** array;

Array in JS can contain any types.

To get length, arrayName.length.

Arrays can also be **multi-dimensional**.

Arrays

var name = [“stuff”, “stuff” ]

can contain different types of variables in same array.

also has zero-based indexing.

Nested arrays var name=[[“dog”, 1],[2,4]];

arrays are *mutable*.

**Push()** = Adding value to end of array, returns new length of array

**Pop()** = removing value from end of an array; returns value

**Shift()** = removing first value from array; returns value

**unShift()** = adding first value to array, returns new length of array

note when using these functions, that you can push individual values arrayName.push(“4”,2,5); in rather than pushing an array arrayName.push([“4”,2,5]) because the array will then go into the last index of arrayName.

**Modify Array**

var deletedElements = array.splice(start, deleteCount, item1, item2...)

returns modified array, deleteCount is number of elements you want to **remove**, and item1, etc are elements that you want to **add** to the array.

**Selecting Parts of Array**

array.slice(beginningIndex, endIndex)

returns a new array from items that you selected. Leaves original array untouched.

If no arguments, shallow copy of entire array.

note that new array goes up to but does not include endIndex.

Data structures are useful for not just storing data but for retrieving it easily too, and modifying it too, sort of like a CRUD. To do this with arrays, use **bracket notation**.

Of course arrays must be **mutable** then in Javascript.

Can use ...**Spread Operator** to quickly extract all values from array e.g. when pushing. Also can be used to combine arrays easily.

let fragment = ['to', 'code'];

let sentence = ['learning',...fragment, 'is','fun'];

Can check if element exists using **indexOf(value)** method. Index Of method will return -1 if doesn’t exist, and will return index Number if does exist.

Challenge code for removing subarray if contains element:

function filteredArray(arr, elem) {  
let newArr = [];  
// change code below this line  
for( var i = 0 ; i<arr.length ; i++){  
 if (arr[i].indexOf(elem) === -1){ // so beautiful!!!  
 newArr.push(arr[i]);  
 }  
}  
// change code above this line  
return newArr;  
}

**Objects**

let userData = FCC\_User['followers']  
// userData equals 572

Notice that with bracket notation, we enclosed followers in quotes. This is because the brackets actually allow us to pass a variable in to be evaluated as a property name (hint: keep this in mind for later!). Had we passed followers in without the quotes, the JavaScript engine would have attempted to evaluate it as a variable, and a ReferenceError: followers is not defined would have been thrown.

Bracket notation is very useful because sometimes object properties are not known before runtime or we need to access them in a more dynamic way.

Let's revisit our foods object example one last time. If we wanted to remove the apples key, we can remove it by using the delete keyword like this:

delete foods.apples;

**hasOwnProperty()**

if we just wanted to know if an object has a specific property? JavaScript provides us with two different ways to do this. One uses the hasOwnProperty() method and the other uses the in keyword. If we have an object users with a property of Alan, we could check for its presence in either of the following ways:

users.hasOwnProperty('Alan');  
'Alan' in users;  
// both return true

**for…in statement**

For our users object, this could look like:

for (let user in users) {  
  console.log(user);  
};  
  
// logs:  
Alan  
Jeff  
Sarah  
Ryan

**Object Keys**

contains all the keys stored in an object using the Object.keys(“object\_as\_argument”);

Object Oriented Programming

Objects can have both values and methods (functions) as properties.

let duck = {  
  name: "Aflac",  
  numLegs: 2,  
  sayName: function() {return "The name of this duck is " + duck.name + ".";}  
};  
duck.sayName();  
// Returns "The name of this duck is Aflac."

**This keyword**

If the variable name changes, any code referencing the original name would need to be updated as well. In a short object definition, it isn't a problem, but if an object has many references to its properties there is a greater chance for error.

A way to avoid these issues is with the this keyword:

let duck = {  
  name: "Aflac",  
  numLegs: 2,  
  sayName: function() {return "The name of this duck is " + this.name + ".";}  
};

*this* is a deep topic, and the above example is only one way to use it. In the current context, this refers to the object that the method is associated with: duck.

**Constructor Function**

Constructors are functions that create new objects. They define properties and behaviors that will belong to the new object. Think of them as a blueprint for the creation of new objects.

Here is an example of a constructor:

function Bird() {  
  this.name = "Albert";  
  this.color = "blue";  
  this.numLegs = 2;  
}

This constructor defines a Bird object with properties name, color, and numLegs set to Albert, blue, and 2, respectively.

Constructors follow a few conventions:

* Constructors are defined with a capitalized name to distinguish them from other functions that are not constructors.
* Constructors use the keyword this to set properties of the object they will create. Inside the constructor, this refers to the new object it will create.
* Constructors define properties and behaviors instead of returning a value as other functions might.

**Creating Objects Using Constructor**

let hound = new Dog();

Notice that the new operator is used when calling a constructor. This tells JavaScript to create a new instance of Bird called blueBird. Without the new operator, this inside the constructor would not point to the newly created object, giving unexpected results.

**Passing Arguments to Constructors**

function Bird(name, color) {  
  this.name = name;  
  this.color = color;  
  this.numLegs = 2;  
}

**Verification using instanceof**

Anytime a constructor function creates a new object, that object is said to be an instance of its constructor. JavaScript gives a convenient way to verify this with the instanceof operator. instanceof allows you to compare an object to a constructor, returning true or false based on whether or not that object was created with the constructor. Here's an example:

let Bird = function(name, color) {  
  this.name = name;  
  this.color = color;  
  this.numLegs = 2;  
}  
  
let crow = new Bird("Alexis", "black");  
  
crow instanceof Bird; // => true

**Own Properties**

name and color are called own properties, because they are defined directly on the instance object. That means that any instance of the object each has its own separate copy of these properties.

**Prototype Properties**

Properties that are inherently part of each object. You don’t need to state it in your constructor, which can take up a lot of space.

Since numLegs will probably have the same value for all instances of Bird, you essentially have a duplicated variable numLegs inside each Bird instance.

This may not be an issue when there are only two instances, but imagine if there are millions of instances. That would be a lot of duplicated variables.

A better way is to use Bird’s prototype. The prototype is an object that is shared among ALL instances of Bird. Here's how to add numLegs to the Bird prototype:

Bird.prototype.numLegs = 2;

Now all instances of Bird have the numLegs property.

**Iterate over Own or Prototype Properties**

Here is how you add duck’s own properties to the array ownProps and prototype properties to the array prototypeProps:

let ownProps = [];  
let prototypeProps = [];  
  
for (let property in duck) {  
  if(duck.hasOwnProperty(property)) {  
    ownProps.push(property);  
  } else {  
    prototypeProps.push(property);  
  }  
}  
  
console.log(ownProps); // prints ["name"]  
console.log(prototypeProps); // prints ["numLegs"]

**Constructor Property**

console.log(beagle.constructor === Dog); //prints true

Note that the constructor property is a reference to the constructor function that created the instance.

The advantage of the constructor property is that it's possible to check for this property to find out what kind of object it is.

Note  
Since the constructor property can be overwritten (which will be covered in the next two challenges) it’s generally better to use the instanceof method to check the type of an object.

**Prototype Property Can be Set as an Object**

For more than a small number of prototype properties, you can set the prototype to an object.

Bird.prototype = {  
  numLegs: 2,   
  eat: function() {  
    console.log("nom nom nom");  
  },  
  describe: function() {  
    console.log("My name is " + this.name);  
  }  
};

However, whenever the prototype is set as an object, the constructor property is erased (duck.constructor returns *undefined*). Which means you need to set it in the object.

Dog.prototype = {

constructor: Dog,

numLegs: 2,

eat: function() {

console.log("nom nom nom");

},

describe: function() {

console.log("My name is " + this.name);

}

};

**IsPrototypeOf**

Can show if ObjectName.prototype is prototype of an instance.

Bird.prototype.isPrototypeOf(duck);  
// returns true

**Prototype Chain**

All objects in JavaScript (with a few exceptions) have a prototype. Also, an object’s prototype itself is an object.

function Bird(name) {  
  this.name = name;  
}  
  
typeof Bird.prototype; // => object

Because a prototype is an object, a prototype can have its own prototype! In this case, the prototype of Bird.prototype is Object.prototype:

Object.prototype.isPrototypeOf(Bird.prototype);  
// returns true

How is this useful? You may recall the hasOwnProperty method from a previous challenge:

let duck = new Bird("Donald");  
duck.hasOwnProperty("name"); // => true

The hasOwnProperty method is defined in Object.prototype, which can be accessed by Bird.prototype, which can then be accessed by duck. This is an example of the prototype chain.

In this prototype chain, Bird is the supertype for duck, while duck is the subtype. Object is a supertype for both Bird and duck.

Object is a supertype for all objects in JavaScript. Therefore, any object can use the hasOwnProperty method.

**Introduction to Inheritance**

There's a principle in programming called Don't Repeat Yourself (DRY). The reason repeated code is a problem is because any change requires fixing code in multiple places. This usually means more work for programmers and more room for errors.

Notice in the example below that the describe method is shared by Bird and Dog:

Bird.prototype = {  
  constructor: Bird,  
  describe: function() {  
    console.log("My name is " + this.name);  
  }  
};  
  
Dog.prototype = {  
  constructor: Dog,  
  describe: function() {  
    console.log("My name is " + this.name);  
  }  
};

The describe method is repeated in two places. The code can be edited to follow the DRY principle by creating a supertype (or parent) called Animal:

function Animal() { };  
  
Animal.prototype = {  
  constructor: Animal,   
  describe: function() {  
    console.log("My name is " + this.name);  
  }  
};

Since Animal includes the describe method, you can remove it from Bird and Dog:

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

**Inheritance: Instance of Supertype (parent)**

You already know one way to create an instance of Animal using the new operator:

let animal = new Animal();

There are some disadvantages when using this syntax for inheritance, which are too complex for the scope of this challenge. Instead, here's an alternative approach without those disadvantages:

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

Object.create(obj) creates a new object, and sets obj as the new object's prototype. Recall that the prototype is like the "recipe" for creating an object. By setting the prototype of animal to be Animal's prototype, you are effectively giving the animal instance the same "recipe" as any other instance of Animal.

animal.eat(); // prints "nom nom nom"  
animal instanceof Animal; // => true

**Inheritance: Passing all of the Parent Properties to Child**

Bird.prototype = Object.create(Animal.prototype);

Remember that the prototype is like the "recipe" for creating an object. In a way, the recipe for Bird now includes all the key "ingredients" from Animal.

**Inheritance: Reset an Inherited Constructor Property**

When an object inherits its prototype from another object, it also inherits the supertype's constructor property.

Here's an example:

function Bird() { }  
Bird.prototype = Object.create(Animal.prototype);  
let duck = new Bird();  
duck.constructor // function Animal(){...}

But duck and all instances of Bird should show that they were constructed by Bird and not Animal. To do so, you can manually set Bird's constructor property to the Bird object:

Bird.prototype.constructor = Bird;  
duck.constructor // function Bird(){...}

**Inheritance: Adding Methods after Inheritance**

In addition to what is inherited from Animal, you want to add behavior that is unique to Bird objects. Here, Bird will get a fly() function. Functions are added to Bird's prototype the same way as any constructor function:

Bird.prototype.fly = function() {  
  console.log("I'm flying!");  
};

**Inheritance: Overriding an Inherited Method**

It's possible to override an inherited method. It's done the same way - by adding a method to ChildObject.prototype using the same method name as the one to override.

Here's an example of Bird overriding the eat() method inherited from Animal:

function Animal() { }  
Animal.prototype.eat = function() {  
  return "nom nom nom";  
};  
function Bird() { }  
  
// Inherit all methods from Animal  
Bird.prototype = Object.create(Animal.prototype);  
  
// Bird.eat() overrides Animal.eat()  
Bird.prototype.eat = function() {  
  return "peck peck peck";  
};

If you have an instance let duck = new Bird(); and you call duck.eat(), this is how JavaScript looks for the method on duck’s prototype chain:

1. duck => Is eat() defined here? No.

2. Bird => Is eat() defined here? => Yes. Execute it and stop searching.

3. Animal => eat() is also defined, but JavaScript stopped searching before reaching this level.

4. Object => JavaScript stopped searching before reaching this level.

**Mixin**

However, there are cases when inheritance is not the best solution. Inheritance does not work well for unrelated objects like Bird and Airplane. They can both fly, but a Bird is not a type of Airplane and vice versa.

For unrelated objects, it's better to use mixins.

A mixin allows other objects to use a collection of functions.

let flyMixin = function(obj) {  
  obj.fly = function() {  
    console.log("Flying, wooosh!");  
  }  
};

The flyMixin takes any object and gives it the fly method.

let bird = {  
  name: "Donald",  
  numLegs: 2  
};  
  
let plane = {  
  model: "777",  
  numPassengers: 524  
};  
  
flyMixin(bird);  
flyMixin(plane);

**Closure to Access Private Values**

bird had a public property name. It is considered public because it can be accessed and changed outside of bird's definition.

bird.name = "Duffy";

Therefore, any part of your code can easily change the name of bird to any value. Think about things like passwords and bank accounts being easily changeable by any part of your codebase. That could cause a lot of issues.

The simplest way to make properties private is by creating a variable within the constructor function. This changes the scope of that variable to be within the constructor function versus available globally. This way, the property can only be accessed and changed by methods also within the constructor function.

function Bird() {  
 **let hatchedEgg = 10; // private property**  
  
**this.getHatchedEggCount = function() {** // publicly available method that a bird object can use  
    return hatchedEgg;  
  };  
}  
let ducky = new Bird();  
ducky.getHatchedEggCount(); // returns 10

Here getHachedEggCount is a privileged method, because it has access to the private variable hatchedEgg. This is possible because hatchedEgg is declared in the same context as getHachedEggCount. In JavaScript, a function always has access to the context in which it was created. This is called closure.

**Immediately Invoked Function Expression (IIFE)**

A common pattern in JavaScript is to execute a function as soon as it is declared:

(function () {  
  console.log("Chirp, chirp!");  
})(); // this is an anonymous function expression that executes right away  
// Outputs "Chirp, chirp!" immediately

Note that the function has no name and is not stored in a variable. The two parentheses () at the end of the function expression cause it to be immediately executed or invoked. This pattern is known as an immediately invoked function expression or IIFE. <http://benalman.com/news/2010/11/immediately-invoked-function-expression/>

**IIFE to create Module (to package Mixins together)**

An immediately invoked function expression (IIFE) is often used to group related functionality into a single object or module. For example, an earlier challenge defined two mixins:

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

We can group these mixins into a module as follows:

let motionModule = (function () {  
  return {  
    glideMixin: function (obj) {  
      obj.glide = function() {  
        console.log("Gliding on the water");  
      };  
    },  
    flyMixin: function(obj) {  
      obj.fly = function() {  
        console.log("Flying, wooosh!");  
      };  
    }  
  }  
}) (); // The two parentheses cause the function to be immediately invoked

Note that you have an immediately invoked function expression (IIFE) that returns an object motionModule. This returned object contains all of the mixin behaviors as properties of the object.

The advantage of the module pattern is that all of the motion behaviors can be packaged into a single object that can then be used by other parts of your code. Here is an example using it:

motionModule.glideMixin(duck);  
duck.glide();

**Functional Programming**

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

INPUT -> PROCESS -> OUTPUT

Functional programming is about:

1) Isolated functions - there is no dependence on the state of the program, which includes global variables that are subject to change

2) Pure functions - the same input always gives the same output

3) Functions with limited side effects - any changes, or mutations, to the state of the program outside the function are carefully controlled

But first, let's cover some functional terminology:

**Callbacks** are the functions that are slipped or passed into another function to decide the invocation of that function. You may have seen them passed to other methods, for example in filter, the callback function tells JavaScript the criteria for how to filter an array.

Functions that can be assigned to a variable, passed into another function, or returned from another function just like any other normal value, are called **first class** functions. In JavaScript, all functions are first class functions.

The functions that take a function as an argument, or return a function as a return value are called **higher order** functions.

When the functions are passed in to another function or returned from another function, then those functions which gets passed in or returned can be called a **lambda**.

e.g.

const prepareGreenTea = () => 'greenTea';

const prepareBlackTea = () => 'blackTea';

const getTea = (prepareTea, numOfCups) => {

const teaCups = [];

for(let cups = 1; cups <= numOfCups; cups += 1) {

const teaCup = prepareTea();

teaCups.push(teaCup);

}

return teaCups;

};

const tea4GreenTeamFCC = getTea(prepareGreenTea, 27);

const tea4BlackTeamFCC = getTea(prepareBlackTea, 13);

FB,Gitter,Reddit,Twitter,Medium,new tab,Netflix,YouTube,[NO VINE],GMail,Work mail,Docs,freeCodeCamp,new tab

**Imperative vs Functional Programming**

In English (and many other languages), the imperative tense is used to give commands. Similarly, an imperative style in programming is one that gives the computer a set of statements to perform a task.

Often the statements change the state of the program, like updating global variables. A classic example is writing a for loop that gives exact directions to iterate over the indices of an array.

In contrast, functional programming is a form of declarative programming. You tell the computer what you want done by calling a method or function.

JavaScript offers many predefined methods that handle common tasks so you don't need to write out how the computer should perform them. For example, instead of using the for loop mentioned above, you could call the map method which handles the details of iterating over an array. This helps to avoid semantic errors, like the "Off By One Errors" that were covered in the Debugging section.

Recall that in functional programming, changing or altering things is called mutation, and the outcome is called a side effect. A function, ideally, should be a pure function, meaning that it does not cause any side effects.

**Pass Arguments**

Another principle of functional programming is to always declare your dependencies explicitly. This means if a function depends on a variable or object being present, then pass that variable or object directly into the function as an argument.

There are several good consequences from this principle. The function is easier to test, you know exactly what input it takes, and it won't depend on anything else in your program.

This can give you more confidence when you alter, remove, or add new code. You would know what you can or cannot change and you can see where the potential traps are.

Finally, the function would always produce the same output for the same set of inputs, no matter what part of the code executes it.

​The way to pass an array into function using functional programming is by using the **slice** function.

**Functional Programming Implementing Map Function using This**

****

**Document ready in vanilla Javascript**

document.addEventListener(“DOMContentLoader”, function(){//do stuff});

Closures:

Closures are functions that refer to independent (free) variables (variables used locally, but defined in an enclosing scope). These variables remember the environment in which they were created.

JSON APIs and Ajax

APIs mean Applicaiton Programming Interfaces

Ajax is a technology that updates HTML with data

// code for getMessage button

$(“#getMessage”).on(“click”, function(){

$.getJSON("/json/cats.json", function(json){

$(".message").html(JSON.stringify(json));

});

#(“.message”).html(“message”); // changes the html message in .message

});

JSON is a data format for transferring API data.

{key :value, key:value}

adding JSON data to html variable.

json.forEach(function(){

var keys = Object.keys(val);

html += "<div class = 'cat'>";

keys.forEach(function(key){

html+= "<strong>" + key + "</strong>: " + val[key] + "<br>";

});

html += "</div><br>"

});

rednering images from JSON

html += "<img src = '" + val.imageLink + "' " + "alt='" + val.altText + "'>";

getting location:

if (navigator.geolocation){

navigator.geolocation.getCurrentPosition(function(position){

$("#data").html("latitude: " + position.coords.latitude + "<br>longitude: " + position.coords.longitude);

});

}

**Calling an api**

url = “something.com”

$.ajax({

Type: “GET”,

url:url,

success:function(data){

//do press

}

Javascript Browser Object Model (BOM)

not all same in difference browsers

window.innerHeight

window.innerWidth

[doesn’t incluse scrollbars]

window.open()

window.close()

window.moveTo()

window.resizeTo()

Window.Screen Object

information on user’s screen, don’t need to use window. Prefix.

Props

screen.width

* screen.height
* screen.availWidth, in pixels minus things like Windows Taskbar
* screen.availHeight
* screen.colorDepth, amount of different colours computer resolution, based on hardware
* screen.pixelDepth, same as colorDepth for modern computers

Window.location object

can be written without window prefix

* window.location.href returns the href (URL) of the current page
* window.location.hostname returns the domain name of the web host
* window.location.pathname returns the path and filename of the current page
* window.location.protocol returns the web protocol used (http: or https:)
* window.location.assign(“URL”) loads a new document
* window.location.port returns port number of page, if default 80 for http and 443 for https, browsers will display 0 or nothing.

Window.history

can be written without window prefix.

* history.back returns previous URL
* history.forward loads next URL on history list

Window Navigator

can be written without window prefix

* navigator.cookieEnabled returns true if cookies enabled, else false
* navigator.appName returns application name of browser, Netscape is application name of IE11, Chrome, Firefox and Safari. So doesn’t give that much info.
* Navigator.appCodeName returns app code name which apparently is Mozilla for pretty much everything.
* Navigator.product returns produce name of browser engine. For Mozilla this is Gecko.
* Navigator.appVersion returns version of browser
* Navigator.userAgent returns browser type, OS and 32bit or 64bit.

Warning, don’t use this navigator object to get browser type because people can change it.

* Navigator.platform returns operating system.
* Navigator.language returns browser’s language.
* Navigator.onLine returns true is browser is online.
* Navigator.javaEnabled() returns true if Java is enabled.

Javascript Popup Boxes

all can be written without window prefix

* window.alert(“string”) can be written without window prefix
* window.confirm(“sometext”) will have OK returns true and Cancel returns false,
* window.prompt(“sometext”, “defaultText”) gives a box with some text and a form.
* Line breaks are done using \n characters.

Timing Events

all can be written without window prefix

* window.setTimeout(function, milliseconds) runs function after waiting for milliseconds
* window.clearTimeout(setTimeoutHandle) stops setTimeout if function has not begin running yet.
* Window.setInterval(function,milliseconds) function is executed between time intervals set by second argument.
* Window.clearInterval(setIntervalHandle) stops setInterval

Cookies

* cookies saved in name-value pairs e.g. username = someone someone
* document.cookie property
* create cookie: document.cookie= “username=John Smith”;
* expiry date of cookie: document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC";
* can tell browser which path cookie belongs to. document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC; path=/";
* reading cookie by: var x= document.cookie
* change cookie same way as creating it.
* Delete cookie by setting expires parameter to passed date.