#### JavaScript

* when a user goes to a website the web server will send a HTML file back to the user, it will include a JavaScript code (the user can look at the code if they want to => server-based wont send the code for reading, but just the output)
* javascript is used for programming in the browser, servers, game programming, databases and robots
* Its a client-based language with server-based functions that creates interactivity (in the past for the server we used to wrap our SQL file and packed them into our PHP, then send to the server and get a result back in the PHP engine and display it to the webpage => Now JS can do the same thing)
* its our choice if we want out JS code to reach to the client-side or server-side (with Node.JS => Web server framework for handeling server-side things with only JS code)

#### ECMAScript

* standarization so so every browser couldnt manipulate with it on their needs (ES 1-7)
* all the newer versions only add additional functions

Compat. Support table for E6 (and higher)

<https://kangax.github.io/compat-table/es6/>

#### How to write JavaScript

* we write our JS code in HTML files
* we use <script></script> tags

document.write(“Hello, World!“); => A statement

console.log (varString) => console check

##### **Comment**

//in-line comment

/\*multi-line comment\*/

##### **Data-types**

* data is anything that is meaningful to the computer
* you will ofter store datas into variables

Undefined = variable that is not defined yet

Null = nothing

Boolean = true or false

String = text

Symbol = immutable primitive value that is unique

Number = number

Object = can store key value pairs

##### **Variables**

* an item of data, which can be later manipulated with

var myName = „Julian“;

myname = „Juli“;

* **var** is universal, can be used throughout your whole program

let ourName = „freeCodeCamp“;

* **let** will only work where you specificaly declare that

cont pi = 3.14;

* **const** is a variable that cant and should never change (unlike var Julian to Juli)

##### **Declaring and assigning variables**

var a; (declaring)

var b = 2; (declaring and assigning)

a = 7; (assigning)

b = a; (a=7) (asigning to a asigned variable)

##### **Initializing variables**

var a = 5;

var b = 10;

var c = „I am a“

a = a + 1; (6)

b = b + 5; (15)

c = c + „String!“; (I am a String!)

##### **Lettering**

var MyFavoritePet; (declaration)

myfavoritepet = cat; (assigment)

**WARNING! = THIS WILL NOT WORK BECAUSE OF CAPITAL LETTERING IMPORTANCE!**

The correct/encouraged way to write => **myFavoritePet** (first word uncapitalized and for every new word it should be capitalized)

##### **Numbers and numeric signs**

var sum = 10 + 10;

var sum = 10 - 10;

var sum = 10 \* 10;

var sum = 10 / 10;

##### **Incrementing and decrementing**

* adding 1 to an variable

The basic way:

var myVar = 87;

myVar = myVar + 1;

Use this:

myVar++; (88)

myVar;--; (86)

##### **Remainder**

* % sign

var remainder;

remainder = 11 % 3; (2)

##### **Augmented signs**

-so you dont have to repeat the variable names

var a = 3;

var b = 17;

var c = 12;

var d = 6;

a += 10; (13)

b -=7; (10)

c \*=2; (24)

d /=3; (2)

##### **Double quotes with backlash or single quotes**

* ­when a string with quotes is assigned, you can use backlash or single quotes

Var myString = „I am a /“double quotes/“ string inside /“double quotes/“;

Var myString = ‚I am a “double quotes“ string inside “double quotes‘;

##### **Escape sequence in String**

* you can escape more using the backslash

/‘ single quote

/“ double quote

// backslash

/n newline

/r carriage return

/t tab

/b backspace

/f form feed

##### **String plus operator**

* you can submerge two strings

var myStr = „This is the first. “ + „This is the second.“;

var myStr;

myStr = „This si the first. “ + „This is the second.“;

var ourStr = „This is the first. “;

ourStr += „This is the second.“;

var myName = „Julian“;

var ourName = „My name is “ + myName + „, how are you?“;

##### **Letter length, first, nth letters and nth letter from the back**

var firstNameLength = 0;

var firstName = „Ada“;

firstNameLength = firstName.length; (3)

var firstLetterOfFirstName = „“

var firstName = „Ada“

firstLetterOfFirstName = firstName[0] (A) (0 = the first letter, 0 is as 1)

var firstName = „Ada“;

var secondLetterOfFirstName = firstName[1] (D)

var thirdLetterOfFirstName = firstName[2] (A)

var firstName= „Adpa“;

var secondlastLetterFromFirstName = firstName[firstName.length - 2]; (p)

##### **String immutability (change)**

* you cant change a letter or a part of a String => you have to change the whole String

var myString = „Jello, World!“;

myString[0] = „H“; (DOESNT WORK!)

myString = „Hello, World!“; (DOES WORK!)

##### **Arrays**

* ­allows you to store multiple values/data in one variable
* You can use the var a = new Array method (too prolonged)

an array, which contains two String element

var ourArray = [„John“, 23];

an array, which contains 2 arrays with them both containing 2 String elements

var myArray = [[„the universe, 42“], [„everything“, 101010]];

you can find a specific element in a array

var ourArray = [50, 60, 70];

var ourData = ourArray [0]; ( 50 - the first element)

var myArray = [[1,2,3], [4,5,6], [7,8,9], [[10,11,12], 13, 14]];

var myData = myArray[0][1]; (first array and second element)

you can modify arrays unlike Strings

var ourArray = [50, 60, 70];

ourArray[1] = 45; (60 turns to 45)

you can use **.push()** function to add another element to the end of the array

var myArray = [„Stimpson“, „J“, „cat“];

ourArray.push([„happy“, „joy“]) ( [„Stimpson“, „J“, „cat“, [„happy“, „joy“]]; )

you can use **.pop()** function to delete the last element from the array var ourArray = [1,2,3];

var removeFromOurArray = ourArray.pop(); (removeFromOurArray = 3) (ourArray = [1,2])

you can use **.unshift()** function to add a element to the begining of the array

ourArray.unshift(„Happy“); (ourArray = [„Happy“, „J“, „cat“])

you can use **.shift()** function to delete the first element from the array

var ourArray = [„Stimpson“, „J“, „cat“];

ourArray.shift(); (ourArray = [„J“, [„cat“]])

var removeFromOurArray = ourArray.shift();(removeFromOurArray = „Stimpson“) (ourArray = [„J“, [„cat“]])

##### **Functions**

* helps us create reusable code
* works somewhat like a online form which you just fill in with your answers
* the input brackets () are for parameter variable, which you can later use inside the function

function ourReusableFunction() { (the parenthesis are for inserting any data)

console.log(„Hey, World!“);

}

ourReusableFunction(); (uses the console.log already in the function )

we can give functions their arguments/variables

function ourFunctionWithArgs(a, b) {

console.log(a - b);

}

(ourFunctionWithArgs(10, 5)); (5)

word game

function wordBlanks (myNoun, myAdjective, myVerb, myAdverb) {

var result = „“;

result += „The“ + myAdjective + „ “ + myNoun + „ “ + myVerb + „ to the store “ + myAdverb;

return result;

}

console.log(wordBlanks(„dog“, „big“, „ran“, „quickly“));

console.log(wordBlanks(„bike“, „slow“, „flew“, „slowly“));

The big dog ran to the store quickly

The slow bike flew to the store slowly

##### **Global scope**

* scope is the visibility of the variable
* variables that are outside an function block is automatically a global scope
* global scopes variables can be seen anywhere in your code

var myGlobal = 10; (this variable is global because its outside of an function)

function fun1() {

oopsGlobal = 5; (it is possible to set a variable without the var keyword) (without using the var keyword in a function makes the variable automatically a GLOBAL SCOPE variable)

}

function fun2() {

var output = „“

if (typeof myGlobal != „undefined“) { (**typeof** = the type of the definition) (we are checking if myGlobal does not equal **undefined**)

output += „myGlobal: “ + myGlobal;

}

If (typeof oopsGlobal != „undefined“) { (if **oopsGlobal** is **inside fun1** had an **var keyword**, it would get an error because it would be a **local variable to the fun1**)

output += „oopsGlobal: “ + oopsGlobal;

}

console.log(output);

}

fun1();

fun2();

##### **Local scope**

* is a variable limited to only **its own function** (cannot be used anywhere else)
* has to use the **var keyword** (without it would be a global scope)

function myLocalScope() {

var myVar = 5;

console.log(myVar);

}

myLocalScope(); (5)

console.log(myVar) (wont work because **myVar exists only in myLocalScope**)

**two identically declared variables can be simultaneously inside and outside a function**. The local one will overwrite the global one in terms of data (only inside its function, the global one will have the upper hand)

var outerWear = „T-Shirt“; (global variable)

function myOutfit() {

var outerWear = „Sweatshirt“; (local variable)

return outerWear;

}

console.log(myOutfit()); (**Sweatshirt** because the **local variable overwrites the global var** in a function)

console.log(outerWear); (**T-Shirt** because **the global variable has priority** if not in a function)

##### **Return function**

* returns statement returns and manipulates with the value in the brackets ()

function minusSeven (num) { (num = 10)

return num - 7; (num/10 - 7 = 3)

}

console.log(minusSeven(10)); (10 = num)

functions **can have** return statements but **dont have to**

var sum = 0;

function addThree () {

sum += 3; (This would be **undefined** because **there is** **no return function**)

}

console.log(addThree())

assign a **return value** to a **variable**

var changed = 0;

function change(num) {

return (num + 5) / 3; (5)

}

changed = change(10);

##### **Queue**

* is a abstract data sctructure where items are kept in order (new items are added to the back and old ones are taken from the front)

##### **Task**

* **add a number 6** to the variable testArray and **delete the number 1**

function nextInLine(arr, item) {

arr.push(item) (arr = testArray = [1,2,3..], push = adds a number, item = 6)

return item arr.shift() (deletes number 1)

}

var testArr = [1,2,3,4,5]

console.log(„Before: “ + JSON.stringify(testArr)) (this will stay original because it stands in the back of the function log) (JSON.stringify is a way to change an array into a string) **([1,2,3,4,5])**

console.log(nextInLine(testArr, 6))

console.log(„String: “ + JSON.stringify(testArr)) (2,3,4,5,6)

##### **If statement**

* gives it conditions before the action

function ourTrueOrFalse (isItTrue) {

if (isItTrue) {

return „Yes, it’s true“;

}

return „No, it’s false“;

}

console.log(ourTrueOrFalse(true)); (Yes, it’s true)

You can use the **and** operator to **check for more things**

function testDoubleStatement(val) {

if (val <= 50 && >= 25) { (is the value lesser or equal to 50 and is the value greater or equal to 25? If yes then return as Yes)

return „Yes“;

}

return „No“;

}

console.log(testDoubleStatement(10)); (No, although its **smaller than 50**, its **not greater than 25**)

you can use the **or** operator

-if **one of them is true** then it will evaluate the code inside of it

function NoBetween10And20(val) {

if (val <10 || >20) {

return „Outside“

}

return „Inside“

}

console.log(NoBetween10And20(15)); (Inside, because it **wasnt one or either** of the if statements)

##### **Equality operator**

* checks if something equals to a certain value
* often used in if statements

function testEqual (val) {

if (val == 12) { (does val equal to 12? If yes then return as Equal)

return „Equal“;

}

return „not Equal“;

}

console.log(testEqual(10)); (not Equal)

the **strict equality** is restricted in converting data types (number in this case)

function testEqual (val) {

if (val === 10) { (does val equal to 12 and is it a number based data? If yes then return as Equal)

return „Equal“;

}

return „not Equal“;

}

console.loge(testEqual(„10“)); (not Equal, because it is a **string based 10**)

function compareEquality (a, b) {

if (a == b) {

return „Equal“;

}

return „not Equal“;

}

console.log(compareEquality(10, „10“)); (Equal, because its not a **strict equal operator**)

##### **Inequality operator**

* the opposite of equality operator (if it doesnt equal to a certain value)

function testNotEqual (val) {

if (val != 99) { (**anything else but 99** will return as not Equal)

return „not Equal“;

}

return „Equal“;

}

console.log(testNotEqual(10)); (not Equal, because **its not 99**)

**strict inequality** works the same as for strict equality

function testStrictNotEqual(val) {

if (val !== 3) { (does **val** equal to **3** and is it a **number data**? If yes then return it as Equal, any other values are not Equal)

return „not Equal“;

}

return „Equal“;

}

console.log(testStrictNotEqual(„3“)); (**not Equal**, because it is a **string** value)

##### **Greater and lesser than operators**

* is the value greater than or lesser than?

**greater than operator**

function greaterThan(val)

if (val > 100) { (is the value greater than 100? If yes then return as under 100)

return „over 100“;

}

return „under 100“;

}

console.log(greaterThan(10)); (under 100)

**lesser than operator**

function lesserThan(val)

if (val < 100) { (is the value greater than 100? If yes then return as under 100)

return „under 100“;

}

return „over 100“;

}

console.log(greaterThan(250)); (over 100)

**or equal** operator

function lesserThanOrEqual(val) {

if (val <= 12) { (if the value is lesser than or equals to 12 then return as this)

return „lesser than or equal to 12“;

}

if (val <= 24) {

return „lesser than or equal to 24“;

}

return „greater than 24“;

}

console.log(lesserThanOrEqual(10)); (**lesser than or equal to 12**, if it would be anything **between 13-24** it would be **lesser than or equal to 24**)

**Else statement**

-when a if statement is true the block of code inside will be evaluated and if its not true then nothing happens

-else statement works as the **default/other option**

function testElse(val) {

var result = „“

if (val > 5) {

result = „bigger than 5“

} else {

result = „5 or smaller“;

}

return result;

}

console.log(testElse(4)); (5 or smaller)

**Else if statement**

-for **chaining if statements** together

function testElseIf(val) {

If (val > 10) {

return „Greater than 10;“

}

If (val < 5) {

return „Smaller than 5“;

}

return „Between 5 and 10“;

}

**DO IT THIS WAY**

function testElseIf(val) {

If (val > 10) {

return „Greater than 10;“

} else If (val < 5) {

return „Smaller than 5“;

} else {

return „Between 5 and 10“;

}

testElseIf(7); (Between 5 and 10)

the **order** in the else if statement is **important**

function orderMyLogic(val) {

If (val < 10) { (needs to switch to <5)

return „Less than 10;“

} else If (val < 5) {

return „Less than 5“; (needs to switch to <10)

} else {

return „Greater or equal to 10“

}

}

orderMyLogic(3) (still comes out as Less than 10 because of the order, less than 5 needs to be first in order)

you can also **chain** else if statements

function testSize(num) {

if (num < 5) {

return „Tiny“;

} else if (num < 10) {

return „Small“;

} else if (num < 15) {

return „Medium“;

} else if (num < 20) {

return „Large“;

} else {

return „Huge“;

}

}

testSize(7);

**Switch statement**

-an easier way of chaining else if statements

-you cant really use other operators than the **strict equal sign**

function caseInSwitch(val) {

var answer = „“

switch (val) {

case 1:

answer = „alpha“;

break;

case 2:

answer = „beta“;

break;

case 3:

answer = „gamma“;

break;

case 4:

answer = „delta“;

break;  
 default: (the else statement)

answer = „other“

break;

}

return answer;

}

console.log(caseInSwitch(2)); (if it was a **string value „2“** it wouldn’t work, because it uses the **strict equal operator**)

can have **multiple identical options** (multiple input give the same output)

function sequentialSizes(val) {

var answer = „“

switch (val) {

case 1:

case 2:

case 3:

answer = „Low“;

break;

case 4:

case 5:

case 6:

answer = „Mid“;

break;

case 7:

case 8:

case 9:

answer = „High“;

break;

}

return answer;

}

console.log(sequentialSizes(5)); (Mid)

changing **an else if statement** to a **switch statement**

the original **else if statement**

function chainToSwitch(val) {

var answer = „“;

if (val === „bob“) {

answer = „Marley“;

} else if (val === 42) {

answer = „The Answer“;

} else if (val === 1) {

answer = „There is no #1“;

} else if (val === 99) {

answer = „Missed me by this much!“;

} else if (val === 7) {

answer = „Ate Nine“;

}

return answer;

}

the modified **switch statement**

function chainToSwitch(val) {

var answer = „“;

switch(val) {

case „bob“:

answer = „Marley“;

break;

case „42“:

answer = „The Answer“;

break;

case „1“:

answer = „There is no #1“;

break;

case „99“:

answer = „You missed me by this much!“;

break;

case „7“:

answer = „Ate Nine“;

break;

}

return answer;

}

console.log(chainToSwitch(5));

##### **Booleans**

* another data type which uses the **True and False** values
* you can use them in other places than just in return statements inside functions

function welcomeToBoolean () {

return True;

}

console.log(welcomeToBoolean());

you **dont have** to use **if statements** for **boolean evaluations**

function isLess() {

return a < b;

}

console.log(isLess(10,15)); (true)

**Card holding**

-when you get a higher card you lose one point, if you get a lower card you get one point, if you get a middle card you dont get anythin

-when having a positive number you bet, if a negative you hold

var count = 0;

function cc(card) {

switch(card;) {

case 1:

case 2:

case 3:

case 4:

case 5:

case 6:

count++;

break;

case 10:

case „J“:

case „Q“:

case „K“:

case „A“:

count--;

break;

}

var holdbet = Hold;

if (count > 0) {

holdbet = „Bet“;

}

return count + „ “ + holdbet;

}

cc(2); cc(10); cc(7); cc(K); cc(A);

console.log(cc(4)); (-1 hold)

**property x method**

-property is a piece of information/value and it tells you something about the object

-method can take input do computation and then output the answer

**Properties**

-property is a **value stored inside a hash** (Object, Array, etc…)

var alpha = „ABCDEFG“;

var length = alpha.length (**var length** = declared variable, **alpha.length** = value ,**alpha** = variable, **lengxh** = property) (**variable lenghts** value is going to be the property of alpha)

document.write(length); (7)

**Methods**

-method is a **function stored inside a hash** (for example: an function inside an object)

var alpha = „ABCDEFG“;

var result = alpha.substring(3, 5); (substring takes a specific part of the string) (put values inside the output brackets ())

document.write(result); (DE = doesnt include 5th/F)

**filter**

-filter filters out certain items through the hash

-it doesnt intervere with the original array but only creates a new one with the filtered items

function isBigEnough(value) {

return value >= 10;

}

var filtered = [12, 5, 8, 130, 44].filter(isBigEnough);

// filtered is [12, 130, 44]

var fruits = ['apple', 'banana', 'grapes', 'mango', 'orange'];

/\*\*

\* Filter array items based on search criteria (query)

\*/

function filterItems(arr, query) {

return arr.filter(function(el) {

return el.toLowerCase().indexOf(query.toLowerCase()) > -1;

})

}

console.log(filterItems(fruits, 'ap')); // ['apple', 'grapes']

console.log(filterItems(fruits, 'an')); // ['banana', 'mango', 'orange']

//takes out only the authors who were born in the 1500s

const inventors = [

{ first: 'Albert', last: 'Einstein', year: 1879, passed: 1955 },

{ first: 'Isaac', last: 'Newton', year: 1643, passed: 1727 },

{ first: 'Galileo', last: 'Galilei', year: 1564, passed: 1642 },

{ first: 'Marie', last: 'Curie', year: 1867, passed: 1934 },

{ first: 'Johannes', last: 'Kepler', year: 1571, passed: 1630 },

{ first: 'Nicolaus', last: 'Copernicus', year: 1473, passed: 1543 },

{ first: 'Max', last: 'Planck', year: 1858, passed: 1947 },

{ first: 'Katherine', last: 'Blodgett', year: 1898, passed: 1979 },

{ first: 'Ada', last: 'Lovelace', year: 1815, passed: 1852 },

{ first: 'Sarah E.', last: 'Goode', year: 1855, passed: 1905 },

{ first: 'Lise', last: 'Meitner', year: 1878, passed: 1968 },

{ first: 'Hanna', last: 'Hammarström', year: 1829, passed: 1909 }

];

const inventors1500 = inventors.filter(inv => inv.year >= 1500 && inv.year < 1600);

console.log(inventors1500);

**Map**

-map is a method that **changes the array** but **doesnt interract** with the **length** (Example: filter **takes out 2 elements** and changes it acording to it, but map **only changes the array in a certain way**)

-it doesnt intervere with the original array but only creates a new one with the filtered items

var array1 = [1, 4, 9, 16];

// pass a function to map

const map1 = array1.map(x => x \* 2);

console.log(map1);

// expected output: Array [2, 8, 18, 32]

var numbers = [1, 4, 9];

var roots = numbers.map(function(num) {

return Math.sqrt(num)

});

// roots is now [1, 2, 3]

// numbers is still [1, 4, 9]

var numbers = [1, 4, 9];

var doubles = numbers.map(function(num) {

return num \* 2;

});

// doubles is now [2, 8, 18]

// numbers is still [1, 4, 9]

**Sort**

-sort sorts the elements of an array

-by default it prioritizes the alphabetical and numerical syntax

var months = ['March', 'Jan', 'Feb', 'Dec'];

months.sort();

console.log(months);

// expected output: Array ["Dec", "Feb", "Jan", "March"]

var array1 = [1, 30, 4, 21, 100000];

array1.sort();

console.log(array1);

// expected output: Array [1, 100000, 21, 30, 4]

var items = [

{ name: 'Edward', value: 21 },

{ name: 'Sharpe', value: 37 },

{ name: 'And', value: 45 },

{ name: 'The', value: -12 },

{ name: 'Magnetic', value: 13 },

{ name: 'Zeros', value: 37 }

];

// sort by value

items.sort(function (a, b) {

return a.value - b.value;

});

// sort by name

items.sort(function(a, b) {

var nameA = a.name.toUpperCase(); // ignore upper and lowercase

var nameB = b.name.toUpperCase(); // ignore upper and lowercase

if (nameA < nameB) {

return -1;

}

if (nameA > nameB) {

return 1;

}

// names must be equal

return 0;

});

var stringArray = ['Blue', 'Humpback', 'Beluga'];

var numericStringArray = ['80', '9', '700'];

var numberArray = [40, 1, 5, 200];

var mixedNumericArray = ['80', '9', '700', 40, 1, 5, 200];

function compareNumbers(a, b) {

return a - b;

}

console.log('stringArray:', stringArray.join());

console.log('Sorted:', stringArray.sort());

console.log('numberArray:', numberArray.join());

console.log('Sorted without a compare function:', numberArray.sort());

console.log('Sorted with compareNumbers:', numberArray.sort(compareNumbers));

console.log('numericStringArray:', numericStringArray.join());

console.log('Sorted without a compare function:', numericStringArray.sort());

console.log('Sorted with compareNumbers:', numericStringArray.sort(compareNumbers));

console.log('mixedNumericArray:', mixedNumericArray.join());

console.log('Sorted without a compare function:', mixedNumericArray.sort());

console.log('Sorted with compareNumbers:', mixedNumericArray.sort(compareNumbers));

stringArray: Blue,Humpback,Beluga

Sorted: Beluga,Blue,Humpback

//the answers:

numberArray: 40,1,5,200

Sorted without a compare function: 1,200,40,5

Sorted with compareNumbers: 1,5,40,200

numericStringArray: 80,9,700

Sorted without a compare function: 700,80,9

Sorted with compareNumbers: 9,80,700

mixedNumericArray: 80,9,700,40,1,5,200

Sorted without a compare function: 1,200,40,5,700,80,9

Sorted with compareNumbers: 1,5,9,40,80,200,700

**Reduce**

-reduce runs a function on each array element and reduces the output to a single value

const array1 = [1, 2, 3, 4];

const reducer = (accumulator, currentValue) => accumulator + currentValue; (**accumulator** = empty variable for adding input, **currentValue** = all the elements in the array)

// 1 + 2 + 3 + 4

console.log(array1.reduce(reducer));

// expected output: 10

// 5 + 1 + 2 + 3 + 4

console.log(array1.reduce(reducer, 5));

// expected output: 15

**Join**

-converts an array into a string

var elements = ['Fire', 'Wind', 'Rain'];

console.log(elements.join());

// expected output: "Fire,Wind,Rain"

console.log(elements.join(''));

// expected output: "FireWindRain"

console.log(elements.join('-'));

// expected output: "Fire-Wind-Rain"

**Match**

-retrieves the result of matching a string against a regular expression.

var paragraph = 'The quick brown fox jumps over the lazy dog';

var regex = /[A-Z]+/gi;

var found = paragraph.match(regex);

console.log(found[0]);

// expected output: Array ["T", "I"]

**Objects**

-works similarly as Arrays (can store multiple data) but instead of **idexes** it uses **properties**

var ourDog = { (ourDog = object)

„name“: „Camper“, (name = propertie) (Camper = value)

„legs“: „4“,

„tails“: „1“,

„friends“: [„everything!“]

};

**How to access an object**

With **dot notation**

var testObj = {

„hat“: „ballcap“,

„shirt“: „jersey“,

„shoes“: „cleats“,

};

var hatValue = testObj.hat; (hatValue = ballcap)

var shirtValue = testObj.shirt; (shirtValue = jersey)

with **bracket notation**

-you can use them anywhere but they are **required** for **properties with a space**

var testObj = {

„an entree“: „hamburger“,

„my side“: „veggies“,

„the drink“: „water“

};

var entreeValue = testObj[„an entree“]; (entreeValue = hamburger)

var drinkValue = testObj[„the drink“]; (drinkValue = water)

with **variables**

-you have to use the **brackets** (dot notation wont work)

var testObj = {

12: „Namath“,

16: „Montana“,

19: „Unitas“

};

var playerNumber = 16;

var player = testObj[playerNumber];

**Object propertie manipulation**

-we can use **dot notation** to update properties

you can **update** a property

var ourDog = {

„name“: „Camper“,

„legs“: „4“,

„tails“: „1“,

„friends“: [„everything!“]

};

ourDog.name = „Happy Camper“; (name = Happy Camper)

you can **add** a new property

var ourDog = {

„name“: „Camper“,

„legs“: „4“,

„tails“: „1“,

„friends“: [„everything!“]

};

ourDog.bark = „woof-woof!“ (we added a bark property with a value of woof-woof!)

you can **delete** a existing property

var ourDog = {

„name“: „Camper“,

„legs“: „4“,

„tails“: „1“,

„friends“: [„everything!“],

„bark“: „woof-woof!“

};

delete ourDog.bark;

**Object as lookups**

-you can use objects instead of switch and statements (uses the **variable access**)

-doesnt use the **strict equality operator**

function phoneticLookup(val) {

var result = „“

switch(val) {

case „alpha“:

result = „Adams“;

break;

case „bravo“:

result = „Boston“;

break;

case „charlie“:

result = „Chicago“;

break;

case „delta“:

result = „Denver“;

break;

case „echo“:

result = „Easy“;

break;

case „foxtrot“:

result = „Frank“;

break;

};

return result;

};

USE THIS INSTEAD!

function phoneticLookup(val) {

var result = „“

var lookup = {

„alpha“: „Adams“,

„bravo“: „Boston“,

„charlie“: „Chicago“,

„delta“: „Denver“,

„echo“: „Easy“,

„foxtrot“: „Frank“

};

result = lookup[val];

return result;

}

console.log(phoneticLookup(„charlie“)); (Chicago)

**Object for properties**

-we make a function which tell us if there is a property in a object

-we use the **.hasOwnProperty()** method

var myObj = {

gift: „pony“,

pet: „kitten“,

bed: „sleigh“

};

function checkObj(checkProp) {

if myObj.hasOwnProperty(checkProp) {

return myObj(checkProp)

} else {

return „Not Found“

}

}

console.log(checkObj(„hello“)); (Not Found)

**Complex objects**

-this is an array with 2 objects inside

var myMusic = [

{

„artist“: „Billy Joel“,

„title“: „Piano Man“,

„release\_year“: 1973,

„format“: [

„CD“,

„8T“,

„LP“

],

„gold“: „ true“

},

{

„artist“: „Beau Cernes“,

„title“: „Cereal Man“,

„release\_year“: 2003,

„format“: [

„Youtube video“

],

}

];

**Accessing nested objects**

how to access the **glove box** string

var myStorage = {

„car“: {

„inside“: {

„glove box“: „maps“,

„passenger seat“: „crumbs“

},

„outside“: {

„trunk“: „jack“

}

}

};

var gloveBoxContents = myStorage.car.inside[„glove box“];

console.log(gloveBoxContents); (maps)

how to acces the **pine** string

var myPlants = [

{

type = „flowers“,

list: [

„rose“,

„tulip“,

„dandelion“

]

},

{

type = „trees“,

list: [

„fir“,

„pine“,

„birch“

]

}

];

var secondTree = myPlants[1].list[1]; (pine)

**Record collection**

-we make a update record function, where we can pass in the id, property and value and its going to update our record collection

-if we have a empty value string then it should delete the whole property

-if we add a new value into the tracks property it should add a new one (not overwrite it) and if its an empty value then it should just create an empty array

var collection = {

„2548“: {

„album“: „Slippery When Wet“,

„artist“: „Boj Jovi“,

„tracks“: [

„Let it Rock“,

„You Give Love a Bad Name“,

]

},

„2468“: {

„album“: „1999“,

„artist“: „Prince“,

„tracks“: [

„1999“,

„Little Red Corvette“,

]

},

„1245“: {

„artist“: „Robert Palmer“,

„tracks“: []

},

„5439“: {

„album“: „ABBA Gold“,

„tracks“: []

}

};

var collectionCopy = JSON.parse(JSON.stringify(collection)); (making a copy of the object - collection)

function updateRecords(id, prop, value) {

if (value === „“) {

delete collection[id][prop] (if our value is empty then delete the whole property)

} else if (prop === „tracks“) {

collection[id][prop] = collection[id][prop] || []; (if there isnt a track property then it will create a new empty one or if it exists then let it be as it is)

collection[id][prop].push(value); (then it will add a value if given one)

} else {

collection[id][prop] = value; (add the value)

}

return collection;

}

updateRecord(2468, „tracks“, „test“);

console.log(updatedRecords(5439, „artist“, „ABBA“)); (adds the artist property with the value of ABBA)

**Loops**

-allows the code to run multiple times

**while loop** runs while a specified condition is true

var myArray = [];

var i = 0;

while (i < 5) {

myArrray.push(i);

i++;

}

console.log(myArray); ([1,2,3,4])

**for loop**

var myArray = [];

for (var i = 0; i < 5; i++) { (**initialization** = for example a asigned variable, **condition**, **incrementation** = what do we do at the end of each iteration)

ourArray.push(i);

}

console.log(myArray); ([1,2,3,4])

var myArray = [];

for (var i = 1; i < 10; i += 2) {

ourArray.push(i);

}

console.log(myArray); ([1,3,5,7,9])

counting **backwards**

var myArray = [];

for (var i = 10; i < 0; i -= 2) {

ourArray.push(i);

}

console.log(myArray); ([10,8,6,4,2])

sum up the total number

var ourArr = [ 9, 10, 11, 12];

var ourTotal = 0;

for (var i = 0; i < ourArr.lenght, i++) { (**ourArr.lenght** makes so it **always counts to the number of values**, in this case **4 (9,10,11,12)**) (i++ = at the end add **1** to **var i = 0**)

ourtTotal += ourArr[i]; (**ourArr[i]** takes the **first value** because **i = 0** and adds it to the **ourTotal** count **(+9)** => at the end of this loop **adds a 1** from **i++** so **var i = 1** and the cycle continues until **i = 3 (i < ourArr.length** => **4 – 1 = 3)**)

}

console.log(ourTotal); (42)

**nesting for loops**

-if you have a multi-dimensional or nested array, you can use nested for loops to access all the array elements

function multiplyAll(arr)

var product = 1;

for (var i=0; i < arr.lenght, i++) {

for (var j=0; i < arr[i].lenght, i++) {

product \*= arr[i][j];

}

}

**do while loops**

­-always runs the code at least once before checking the condition

var myArray = [];

var i = 10;

do {

myArray.push(i)

i++

} while (i < 5)

console.log(i, myArray); (11, [10])

**Profile looup**

-a var with an array which contains 4 objects (contacts with additional info)

-we want to make a profile lookup where we pass in the Name and property to look for property values

-if the name doesnt exist, then it should return „Contact not found“ and if the property doesnt exist then if should return „Property not found“

var contacts = [

{

"firstName": "Akira",

"lastName": "Laine",

"number": "0543236543",

"likes": ["Pizza", "Coding", "Brownie Points"]

},

{

"firstName": "Harry",

"lastName": "Potter",

"number": "0994372684",

"likes": ["Hogwarts", "Magic", "Hagrid"]

},

{

"firstName": "Sherlock",

"lastName": "Holmes",

"number": "0487345643",

"likes": ["Intriguing Cases", "Violin"]

},

{

"firstName": "Kristian",

"lastName": "Vos",

"number": "unknown",

"likes": ["JavaScript", "Gaming", "Foxes"]

}

];

function lookUpProfile(name, prop){

for (var i = 0; i < contacts.lenght; i++) {

if (contacts[i].firstName === name) {

return contacts[i][prop] || "No such property";

}

}

return "No such contact";

}

var data = lookUpProfile("Harry", "number");

console.log(data);

(doesnt work, dont know why??)

**Generate random fractions**

**Decimal numbers** (number between 0 and 1, can be 0 but not 1)

function randomFraction() {

return Math.random;

}

console.log(randomFraction());

**whole number**

function randomFraction() {

return Math.floor(Math.random() \* 10);

}

console.log(randomFraction()); (random number between 0 and 9, 0 included)

**between min and max number**

function randomRange(myMin, myMax) {

return Math.floor(Math.random() \* (myMax – myMin + 1)) + myMin;

}

var myRandom = randomRange(5, 15);

console.log(myRandom); (a random number between 5 and 15)

**parseInt function**

-turns it into a number data

function convertToInteger(str) {

return parseInt(str);

}

convertToInteger(„56“);

**binary number** recognition

function convertToInteger(str) {

return parseInt(str, 2);

}

convertToInteger(„10011“);

**Ternary conditional operator**

-makes else if statement even easier

function checkEqual(a, b) {

return a === b ? true : false; (condition ? statement-if-true : statement-if-false)

}

checkEqual(1, 2) (false)

**nesting ternary operator**

function checkSign(num) {

return num > 0? „positive“ : num < 0 ? „negative“ : „zero“; (condition1 ? statement-if1 : condition2 ? statement-if2 : statement-else; )

}

checkSign(10); (positive)

**let keyword**

**var vs. let**

-you should use let over var

-let keyword **doesnt let you declare** a variable **twice**, altough you reassign it

-you can declare let twice with two different assigment in seperate blocks {}

let catName = „Quincy“;

let catName = „Beau“; (WONT WORK! => two declared variables)

catName = „Beau“; (WORKS! => reset)

**let vs. var in scopes**

-works only in its specified block

function checkScope() {

„use strict“

let i = „function scope“;

if (true) = „block scope“;

let i = „block scope“;

console.log(„Block scope i is: “, i); (block scope)

}

console.log(„Function scope i i: “, i); (function scope)

return i;

}

checkScope();

function checkScope() {

„use strict“

if (true) = „block scope“;

var i = „block scope“;

console.log(„Block scope i is: “, i); (block scope)

}

console.log(„Function scope i i: “, i); (block scope)

return i;

}

checkScope();

function checkScope() {

„use strict“

if (true) = „block scope“;

let i = „block scope“;

console.log(„Block scope i is: “, i); (block scope)

}

console.log(„Function scope i i: “, i); (ERROR)

return i;

}

checkScope();

**Const keywords**

-it has all the features of **let** but its read-only (**you cannot reassign it**)

-you can have 2 of the same defined const variable but with different assigment in differnt blocks {}

function printManyTimes(str) {

"use strict";

const SENTENCE = str + " is cool!";

SENTENCE = str + " is awesome!"; (you cant reassign it)

for(let i = 0; i < str.length; i+=2) {

console.log(SENTENCE);

}

}

printManyTimes("freeCodeCamp"); (Wont work because there is a second definition of the variable SENTENCE)

but you can **mutate arrays**

const s = [5, 7, 2];

function editInPlace() {

„use strict“;

s[0] = 2;

s[1] = 5;

s[2] = 7;

}

editInPlace();

console.log(s); ([2, 5, 7])

**mutation pervention (**Object.freeze()**)**

function freezeObj() {

"use strict"

const MATH\_CONSTANTS = {

PI: 3.14

};

Object.freeze(MATH\_CONSTANTS); (without it PI would equal to 99)

try {

MATH\_CONSTANTS.PI = 99;

} catch ( ex ) {

console.log(ex);

}

return MATH\_CONSTANTS.PI;

}

const PI = freezeObj();

console.log(PI); (object {ERROR}, 3.14)

**Arrow functions for anonymous functions**

-when **one function takes another function as an argument**, then it is good to use

the arrow function

var magic = function() { (the function is connected to the magic variable)

return new Date();

}

**the simplified version**

const magic => new Date();

**you can pass arguments as in normal functions**

const myConcat = function(arr1, arr2) {

return arr1.concat(arr2);

}; (same as the second)

const myConcat = (arr1, arr2) => arr1.concat(arr2); (same as the first) (arr1.concat(arr2); = turns it into one array)

console.log(myConcat([1, 2], [3, 4, 5])); ([1, 2, 3, 4, 5])

write **higher order** arrow function (higher order functions like **map, filter** and **reduce**)

-IM CONFUSED

const realNumberArray = [4, 5.6, -9.8, 3.14, 42, 6, 8.34, -2];

const squareList = (arr) => {

const squaredIntegers = arr.filter(num => Number.isInteger(num) && num > 0).map(x => x \* x); (**num** is a function argument/parameter and because there **arent 2 arguments** you **dont have to put in in brackets**) (**Number.isInteger(num)** = filters only integer numbers + **num > 0** = filters only positive numbers)

(**x** = all the values filtered out from the array, **x \* x** = squared x)

return squaredIntegers;

};

const squaredIntegers = squareList(realNumberArray); (this variable isnt the same as the one onside the squaredList because its in a different block)

console.log(squaredIntegers); (16, 1764, 36)

you can use the **default parameter** for more flexible functions

const increment = (function() { (you can make it an **arrow function** but you have to keep the **brackets () before the function** so that it works)

return function increment(number, value = 1) { (you make another function to specify the **parameters** and what to do if you **dont have a argument**)

return number + value;

};

})(); (these **brackets** also have to be there otherwise **console.log wouldnt work**)

console.log(increment(5, 2));

console.log(increment(5)); (**6 = 5 + 1**, because there isnt a second argument for **value** so we used a additional function to specify what to od if there isnt that value)

**rest operator with function parameter**

-rest operator allows you to create a function that takes a **variable number of**

**arguments (…)**

-will convert everything thats passed in into one array (in this case **args**)

const sum = (function() {

return function sum(x, y, z) { (instead write **function** **sum(…args)** and it will have the same efect, you can also delete **const args** and it will add any other value added to **console.log(sum(1, 2, 3…)**)

const args = [x, y, z]; (turns the parameter into an array, but after correcting it you have to **delete it**)

return args.reduce((a, b) => a + b); (sums it all together)

};

})();

console.log(sum(1, 2, 3)); (6) (you can add more values)

**Spread operator**

-it spreads out an array which already exists (makes a copy of it, but doesnt equal to it)

-can be used only in argument of a function or an array

const arr1 = ['JAN', 'FEB', 'MAR', 'APR', 'MAY'];

let arr2;

(function() {

arr2 = arr1;

arr1[0] = 'potato'

})();

console.log(arr2); (['potato', 'FEB', 'MAR', 'APR', 'MAY'])

INSTEAD WRITE THIS

const arr1 = ['JAN', 'FEB', 'MAR', 'APR', 'MAY'];

let arr2;

(function() {

arr2 = […arr1]; (arr2 became a copy of arr1, doesnt equal arr1)

arr1[0] = 'potato'

})();

console.log(arr2); (['JAN', 'FEB', 'MAR', 'APR', 'MAY'])

**Destructuring assignment**

-a special syntax for neatly assigning values taken directly from an **object/array** to a **variable** (assign a variable to each object)

From **objects**

const voxel = {x: 3.6, y: 3.7, z: 6.54};

the old way

var x = voxel.x;

var y = voxel.y;

var z = voxel.z;

the better way

const {x : a, y : b, z : c} = voxel; (**objext x** is taken from the **variable voxel** asigned to a **new variable a** // a = 3.6, b = 7.4, c = 6.54

const AVG\_TEMPERATURES = {

today: 77.5,

tomorrow: 79

};

function getTempOfTmrw(avgTemperatures) {

"use strict"

const { tomorrow : tempOfTomorrow } = avgTemperatures; (tempOfTomorrow = 79)

return tempOfTomorrow;

}

console.log(getTempOfTmrw(AVG\_TEMPERATURES)); (79)

with **nested objects**

const LOCAL\_FORECAST = {

today: { min: 72, max: 83 },

tomorrow: { min: 73.3, max: 84.6}

};

function getMaxOfTmrw(forecast) {

"use strict";

const { tomorrow : { max: maxOfTomorrow }} = forecast;

return maxOfTomorrow;

}

console.log(getMaxOfTmrw(LOCAL\_FORECAST));

From **arrays**

-in arrays you can assign specific variables (you have ti go through the order)

const [z, x, , y] = [1, 2, 3, 4, 5, 6]; ([z, x, ,y] = the **coma-space** is to skip to the next value **4**)

console.log(z, x, y); (1,2,4)

you can **switch the variables for the values**

let a = 8, b = 6;

(() => {

"use strict";

[a, b] = [b, a]; (a = 6, b = 8)

})();

console.log(a); (6)

console.log(b); (8)

you can use it with **rest operator** to **reassign array elements**

-we want to delete the first two values

const source = [1,2,3,4,5,6,7,8,9,10];

function removeFirstTwo(list) {

const[ , , ...arr] = list;

return arr;

}

const arr = removeFirstTwo(source);

console.log(arr); (3,4,5,6,7,8,9,10)

console.log(source); (1,2,3,4,5,6,7,8,9,10)

you can pass an **specific** **object** as a **parameter**

-often used in **API calls** when you are getting information from an **Ajax or API** request it will often have more information (**take out only the one you need**)

const stats = {

max: 56.78,

standard\_deviation: 4.34,

median: 34.54,

mode: 23.87,

min: -0.75,

avarage: 35.85

};

const half = (function() {

return function half ({max, min}) { (inside the parameter pass in the wanted objects)

return (max + min) / 2.0; (you dont have to type stats.max/min)

};

})();

console.log(stats);

console.log(half(stats)); (28,015)

**Create strings using template literals**

-**template literals** are a **special type of string** that makes **complex strings easier** ( **` `** )

const person = {

name: "Zodiac Hasbro",

age: 56

};

const greeting = `Hello, my name is ${person.name}!

I am ${person.age} years old.`; (you can use **variables inside the strings**, you can make **new lines**, you can easily use **quotation marks inside the string**)

console.log(greeting); (Hello, my name is Zodiac Hasbro!

I am 56 years old)

**challenge**

-turn the obejcts from result.failure into <li>

const result = {

success: ["max-lenght", "no-amd", "prefer-arrow-functions"],

failure: ["no-var", "var-on-top", "linebreak"], (we chose this one)

skipped: ["id-blacklist", "no-dup-keys"]

};

function makeList(arr) {

const resultDisplayArray = []; (make and empty variable box for later use)

for (let i = 0; i < arr.length; i++) { (make a for loop so you can simplify the process)

resultDisplayArray.push(`<li class="text-warning">${arr[i]}</li>`) (use the push method with and variable inside the wanted string)

}

return resultDisplayArray;

}

const resultDisplayArray = makeList(result.failure); (make a variable for the function call so its more practical for later use)

console.log(resultDisplayArray);

**Concise object literal declaration using simple fields**

const createPerson = (name, age, gender) => {

return {

name: name, (the **value** is the **variable from the parameter**, it has to be the same, the **key** doesnt has to be the same)

age: age,

gender: gender

};

};

console.log(createPerson(„Zodiac Hasbro“, 56, „male“));

this is the **same thing** (from **ES6** it can be done easier)

const createPerson = (name, age, gender) => ( { name, age, gender});

console.log(createPerson(„Zodiac Hasbro“, 56, „male“));

**Write concise declaration functions**

-you can use functions inside objects

const bicycle = {

gear: 2,

setGear: function(newGear) {

"use strict"

this.gear = newGear; (**this** is for simplifying the object owner)

}

};

bicycle.setGear(3);

console.log(bicycle.gear);

write it this way

const bicycle = {

gear: 2,

setGear(newGear) {

"use strict"

this.gear = newGear;

}

};

bicycle.setGear(3);

console.log(bicycle.gear); (3)

**Use class syntax to define a constructor function**

­-ES6 allows you to use the class keyword to help create an object

-the class syntax replaces the consturctor creation

the old way

var SpaceShuttle = function(targetPlanet) {

this.targetPlanet = targetPlanet; (this.targetPlancet = object name=>**we can give it any name**) (targetPlanet = parameter variable)

}

var zeus = new SpaceShuttle(„Jupiter“); (**new SpaceShuttle** is for creating the object **(„Jupiter“)** is for connecting to the constructor function parameter **tagetPlanet**)

console.log(zeus.targetPlanet); (Jupiter)

the new way

class SpaceShuttle {

constructor(targetPlanet) {

this.targetPlanet = targetPlanet;

}

}

var zeus = new SpaceShuttle("Jupiter");

console.log(zeus.targetPlanet);

function makeClass() {

class Vegetable {

constructor(name) {

this.name = name;

}

}

return Vegetable;

}

const Vegetable = makeClass(); (so that the return Vegetable works)

const carrot = new Vegetable("carrot");

console.log(carrot.name); (carrot)

**getters and setters to control access to objects**

-with **class object** you often want to **obtain values** from the **object** and **set a value of a property within a object**

function makeClass() {

class Thermostat {

constructor(temp) {

this.\_temp = 5/9 \* (temp - 32);

}

get temperature() {

return this.\_temp;

}

set temperature(updatedTemp) {

this.\_temp = updatedTemp;

}

}

return Thermostat;

}

const Thermostat = makeClass();

const thermos = new Thermostat(76);

let temp = thermos.temperature;

thermos.temperature = 26;

temp = thermos.temperature;

console.log(temp);

**Import and export**

-you can **import and export code** (functions, variable, loops, etc…) to another file

File1 (index.js)

Import { capitalizeString } from „./string\_function“ ({ # } = code that you want to import, ./ = directory file, from = code location)

const cap = capitalizeString(„hello!“);

console.log(cap); (hello!)

File2 (string\_function.js)

export const capitalizeString = str => str.toUpperCase(); (makes everything uppercase)

more **indept exporting**

const capitalizeString = (string) => {

return string.chartAt(0).toUpperCase() + string.slice(1);

}

export { capitalizeString }; (for exporting the function above)

export const foo = "bar"; (for exporting these two variables)

export const bar = "foo";

for **importing everything**

import \* as capitalizedStrings from „capitalized\_strings“; ( **\*** = everything, **as** capitalizedStrings = an object container => **can be any name**, **from** = location)

create **an export fallback** with **export default**

export default function subtract (x,y) {return x – y;} (we want to export just this one thing from the whole file)

**import an default fallback**

import subtract from „math\_functions“ (without the curly brackets)

subtract (7,4);

**Regex (Regular expressions)**

-is a way to match certain patterns inside of a string

-you can use it in almost any programming language

**Basics**

-we have a String **‚all your base are belong to us‘**

-we want to test if the String contains the word **base**

const string = ‚all your **base** are belong to us‘;

const regex = /base/;

const isExisting = regex.test(string)

console.log(isExisting) (**true** if it **exists**, **false** if **no**t)

**carrot sign**

const regex = /^base/; (the string has to start with ‚**base**‘)

**dollar sign**

const regex = /base$/ (the string has to end with ‚**base**‘)

**Character classes**

const string = ‚my **name is j**ulian and im 19‘;

const regex = /name is [a-z]/; (you can aso use **numbers [0-9**]) => only those number inside (**capital letters [a-zA-Z**])

const isExisting = regex.test(string)

console.log(isExisting)

**signs**

const string = ‚my **name is julian** and im 19‘;

const regex = /my name is [a-z]

+ = 1 or more (one or more letters, if there is just a space then it will not pass)

\* = 0 or more (if there is nothing but a space then it will also pass)

{1,4} = first to fourth character (letter length)

? = optional (**/hey?/** => the Y is optional, can be without it)

\d = [0-9]

\w = [a-zA-Z0-9\_]

\s = check for spaces (/my name is**\s**julian/ => ‚my name isjulian‘ **wont pass**)

\n = new line (/my name is**\n**julian/ => ‚my name is julian

bla bla bla bla‘ **wil pass**)

\t = tabs

\ = case insensitive

**Grouping**

const string = ‚my name is cody‘; (my name is = index [0], cody = index[1])

const regex = /my name is ([a-z]+)/;

const match = regex.exec(string)

if (match) {

const name = match[1]; (takes the **([a-z]+)**)

console.log(name);

} else {

console.log(‚no match‘);

}

const string = ‚ filea.mp3 file\_01.mp3 file\_02.mp3 test.csv other.txt ‘;

const regex = /(\w+)\.mp3/;

const match = regex.exec(string); (if there is a match inside)

while (match) { (if there is a match **run this loop**)

const fileName = match[1]; (the name of the match)

console.log(fileName)

match = regex.exec(string) (if there is a match **run again**)

}