Begin: Javascript

Chapter 1: Getting Started

Javascript is a programming, scripting and markup languages.

Due to intense community support and investment of big companies, these days with javascript you can build:

- web/ Mobile apps
- real time network apps
- Command line Tools
- Games

Javascript run in the javascript engine of the browser you are using.

- The browsers have a JavaScript Engine (firefox: SpiderMonkey; Chrome:v8)
- Node = v8 + c++
- Ecmascript is specification = reponsible for defining standards or features for javascript

Setting-up the Environment

- An IDE or code editor = Visual studio code.
- Download nodejs
- Live server (in Visual studio code)
- The index.html will be our host for the javascript code.
- Live server template allow us to serve our code into the browser simultaneously.

Where to run JS code?

- Head tag or Body Tag

  It is best practice to put the js codes at the end of the body tag.
  Two main reason:

1. your browser is going to parse the index file (run over the code). ANd since you will have a lot of javascript codes, the browser might get busy by going over those codes. (At the end your web page will take time to load) It won't be able to render the web page. This will create a bad user experience.
2. The code, that we have on this web page, needs to talk to the element of this web page.

- Exception: If you are using third party codes you can put them on the head tag or section.

Basics?

- A statement: is a piece of code that express an action to be carried out. In this case we want to log a message in the console. It ends by a ";".

- A string: is a sequence of character.
- //: is used to add comment.

Separation concern

> Just as Css, you can create a separate .js files (.css and the case of css). Then you will have to link this file to the index.html file or the main html file with an link tag (anchor or hyperlink tag). We want to separate html, which is all about content, from javascript which is all about behavior.

In our case, we have already an index.html. We create an index.js files then we add ou js codes:

```
// comment for the why's
console.log("Hello World");
```

Now we link the index.js to the html like this.

```
<script src="index.js"></script>
```

Chapter 2: Variables

> In programming we use variable to store data temporarily in the computers memory.

-> In JS you have to declare your variable.

> In principle "var" should do the job. But there 's a lot of problem associate to this method. The best practice is to use "let"

```
let name = 'Alexandro Disla';
console.log(name);
// some rules:
// the variable can't be a reserve keywords
// They should be meaningful
// They can't start with a number
// they can't contain space or number
// They are case-sensitive
// stick to camel notation  (FirstName)
```

> best pratice to declare multiple varables

```
let FirstName =  "Alex"
let LastName = "Disla"
```

Constant

-> They are constant variables

```
const irate = 0.3;
irate = 10;
 // it will throw an error
//we can't not reassign a constant
console.log(irate)
```

> Best practice: If you don't need to reassine the value of a variable. Use "const" when you a declaring and initiate the value of the variable.

Primtives and References types

We have two caterogies:

- primitives/value types
- references types

Primitives

1. string
2. number
3. boolean
4. undefined
5. null
6. symbol

```
let name = 'alex'; // string literal
let age = 30; // numeric literal
let isapproved = true; // or false. boolean literal
let fname = undefined; // let fname; will do the same.
// not very common to do let fname = undefined;.
let selectedcolor = null; // to clear the value of a variable. typof -> objects!!!!!
```

Dynamic Typing

> type of a variable is set. Type of a variable change at run time.

- static (statically-typed) language

  Static typing means that types are known and checked for correctness before running your program. This is often done by the lan-

guage's compiler. For example, the following Java method would cause a compile-error, before you run your program:

```
public void foo() {
    int x = 5;
    boolean b = x;
}
```

- dynamic (dynamically-typed) language

  Dynamic typing means that types are only known as your program is running. For example, the following Python (3, if it matters) script can be run without problems:

```
def erroneous():
    s = 'cat' - 1

print('hi!')
```

  It will indeed output hi!. But if we call erroneous:

```
def erroneous():
    s = 'cat' - 1

erroneous()
print('hi!')
# A TypeError will be raised at run-time when erroneous is called.
```

  javascript is a dynamic language

Reference Types

- Object
- Array
- function

Object

when we are dealing with multiple related variables. we can put these variables inside of an object. Objects have properties.

```
let person = {
    name:'Alexandro',
    age: 27
}; // object literal - key:value pairs

// two way to access the properties

// dot notation
person.name = 'Disla';
```

```
// that's the default way

// bracket notation. pass a srting
// with the target property
 person['name'] = 'Sandro';
// this method has his benefit, a user can access
// a target property at run time. Dynamic way!!!
let selection = 'name';
person[selection] = 'Lelex';
// if we did
// let person = {}; we would have an empty array.
 console.log(person.name);
```

Array

The index of the first element in an array start with 0. We use the index to access the elements inside of an array. The fact that javascript is a dynamic language means that the length of an array (position of the elements) change at run time. In javascript we can store different types of element in an array. Technically, An array is an object. Why? The array has multiple properties that we can access. An Array is a data structure that we use to represent a list of items.

```
let selectedColors = ['red', 'blue'];
// array literal
// let selectedColors = []; - empty arrays
selectedColors[2] = 1;

console.log(selectedColors.length)
```

Function and types of function

Function is a fundamental building blocks in javascript. A set of statements that performs a task or calculate a value. The function can have "parameters (at the time of declaration) = arguments (actual value we supply to that parameter)". The defaulf value of variables in JS is undefined. Technically, seems like a function is also an object.

```
// performing a task
function greet(fname,lname){
    // string concatenation similar to
    // python, c++ and c#
    console.log('Hello'+' '+fname+' '+lname);
} // no need of ; function literal
```

```
 greet("Jhonny", "Bravo");

// calculate something
function square(number){
    return number*number;
}

console.log(square(2));
// uesed as an argument of console.log
// square(2) is the expression of the function
// square.
```

Chapter 3: Operators

we have different kind of operator.

> variable or constant + operator = expression.

With this expression:

> expressions -> logic or Algorithms.

list of operators:

- Arithmetic
- Assignement
- Comparison
- Logical
- Bitwise

Arithmetic Operator

```
let x = 10;
let y = 11;

// arithmetic
console.log(x+y);
//x+y is an expression.
// An expression is a piece of code that produce a value.
console.log(x-y);
console.log(x/y);
console.log(x*y);
console.log(x%y); // modulo
console.log(x**y); // exponantiel

// incrementation
console.log(x++); // 10
//another behavior
console.log(++x); // 11
```

```
// decrimentation
console.log(x--); // 10
//another behavior
console.log(--x); // 9
```

Assignement Operator

```
// Assignement operator
let x = 10;
// increment operator
x++;
 // is the same as
 x = x + 1;
 // other possibility
x = x+5;
x+=5;
x = x*3;
x*=3;
// all the arithmetic operator
//  has this shorthand combination pattern
// with the assignment operator
```

comparison operator

```
// comparison operator
let x = 100;
//list of operator

// relational
console.log(x>0);
console.log(x<0);
console.log(x>=0);
console.log(x<=0);
//equality
console.log(x===100);
console.log(x!==100);
//logical
console.log(x>0 && x<101); // a et b
console.log(x>0 || x>100); // a ou b
```

> The result of an expression that include comparison operator is a boolean (true or false).

Equality operator

> The strict equality operator insures to us that both side have the same type and value.
> The lose equality operator doesn't care about the types machine. If the types aren't the same, it will convert the type we have on the right side to match what we have on the left side. And then it will check if the values are equal.

The same reflexion applies to strict inequality or lose inequality.

```
//strict equality
console.log(0===false);

//lose equality operator
console.log(0==false);
// false is convert naturally to 0.
// remember true is 1 and false is 0.

// In console
[Log] false (main.js, line 3)
[Log] true (main.js, line 6)

// analogicly

//strict inequality
console.log(0!==false);

//lose inequality operator
console.log(0!=false);

//strict equality
console.log(0!==false);

//lose equality operator
console.log(0!=false);

//in console
[Log] true (main.js, line 3)
[Log] false (main.js, line 6)
```

In any given situation, The strict way is more precise and accurate.

Ternary operator

```
let p = 110;
let t = p > 100 ? 'gold' : 'silver';
// ? and : are the ternary operator
```

```
// ? 'gold' mean if expression true then gold be display
// : 'silver' if not sylver will be dislplay
console.log(t);
// console
[Log] gold (main.js, line 5)

let p = 3;
let t = p > 100 ? 'gold' : 'silver';
// ? and : are the ternary operator
// ? 'gold' mean if expression true then gold be display
// : 'silver' if not sylver will be dislplay
console.log(t);
//console
[Log] silver (main.js, line 7)
```

logical operator

```
console.log(true&&true);
// is true, only in this case.
// maths a et b (logic matrix)
console.log(true||false);
// is true, if one of the operand is true
// a ou b

console.log(!true);
// will return false
// non a
```

logical operator with non-boolean

The result of a logical expression is not necessarly a true or false. That depends entirely on the value of the operands.

In javascript, we have the concept of falsy or truthy

falsy:

- false
- undefined
- null
- 0
- false
- ” or ””
- NaN

Anything that is not falsy is truthy

```
false||"a"
// javascript will evaluate "a"
```

```
// "a" is not an empty string so
//"a" is truthy and will be return
false||1
// 1 is a truthy and will be return

// short-circuting
false||1||2
// the evaluation will stop right after
// false||1
// 1 will be return
```

Real world example:

```
let usercolor = 'red';
let defaultcolor = 'blue';
let currentColor = usercolor || defaultcolor;

console.log(currentColor);
// in console
[Log] red (main.js, line 5)
////
let usercolor = ''; // or undifined (they are falsy)
let defaultcolor = 'blue';
let currentColor = usercolor || defaultcolor;

console.log(currentColor);
// in console
[Log] blue (main.js, line 5)
```

Bitewise Operator

Decimal TO binary

```
// bitewise OR is different than logical or ||
// bitewise AND is different than logical or &&
console.log(1|2); // R
console.log(1&2); // R2
// decimal to binary
// 1 = 00000001
// 2 = 00000010
// the evaluation process:
// R = 00000011 = 3 // becuz of the OR
//R2 = 00000000 = 0 // becuz of the AND

//in  console
[Log] 3 (main.js, line 3)
[Log] 0 (main.js, line 4)
```

Lets implement a real life example:

```
// Access controle System
// read, write, execute
// 00000100 read  = 4
// 0000010 read and write = 2
// 00000001 execute = 1

const readperm = 4;
const writeperm = 2;
const eperm = 1;

// permission recorded
let myperm = 0;
// we add permission with bitewise OR operator
myperm = myperm | readperm | writeperm | eperm;

// evaluation
// check permission with the bitewise AND operators
let message =(myperm & readperm) ? "yes" : "no";

console.log(message);
// in console
[Log] yes (main.js, line 20)
```

operator precedence

```
// P.E.M.D.A.S

let x = 2+3+4*3;
console.log(x);
// 17
```

EXercise - Chapter 3

Swapping Value of 2 variables

```
let a = "blue";
let b = " red";

let c = " ";

// overwrite logic:
c = a;
a = b;
b= c;
```

```
console.log(a);
console.log(b);
```

Chapter 4: Control Flow

COnditional Statements

IN javascript we have two type of conditional statements:

- if and else
- switch and case

If and Else

```
// basics structure
if(condition){
    // one or multiple
    statements;
}
else if(anothercondition){
    // one or multiple
    statements;
}
// we can have multiple else if
else if(yetanothercondition){
    // one or multiple
    statements;
}
else{
    // if nun of the previous condition are true
    // one or multiple
    statements;
}
// what we call condition here is a mixed of expression and operators.
```

A practical example:

```
let hour = 22;

if(hour>=6&&hour<12){
    console.log('Good morning');
}
else if(hour>=12&&hour<=18){
    console.log('Good Morning');
}
else{
    console.log('Good Evening');
}
```

Switch and case

```
let variable;

switch(variable){
    case 'value':
        statements;
        break;
    // Multiple case
    case 'value2':
        statements;
        break;
    // the default doesn't need a break
    default:
        statements;
}
```

With switch and case we can compare the value of a variable against different value.

Loop

For Loop

we use loop to repeat an action over a number of time. In javascript we have:

- for loops
- while loops
- Do while loops
- for in loops
- for of loops

```
for(initializexpr;condition;incrementexpr){
    // this loop will run as long as the condition will evaluate to true.
    statements;
}
```

Incrementation

```
// i is the loop varaible
for(let i =0;i<5;i++){
    console.log("Hello World");
}

for(let i =0;i<5;i++){
    if(i%2!==0){
        console.log(i);
    }
}
```

Decrementation

```
for(let i=5;i>=0;i--){
    if(i%2!==0){
        console.log(i);
    }
}
```

While Loop

In for loops, the lopp variable is part of the loop itself. In a while loop you have to declare this variable externally.

```
// firt the loop variable
let lvar;
while(conditon){
    statements;
}
```

We will make a while loop equivalent to the previous for loop.

```
let i = 0;
while(i<=5){
    if(i%2!==0){
        console.log(i);
    i++;
    }
}

let i = 5;
while(i>=0){
    if(i%2!==0){
        console.log(i);
    i--;
    }
}
```

DO While loop

Do not make this mistake!

```
let i=0;
while(i<=5){
    if(i%2!==0){
        console.log(i);
    i++;
    }
}
```

```
let i = 0;
// we will get an error in this case. We can't reassign this loop variable.
```

A do while loop:

```
let lvar;
do{
    staements;
}while(condition);
```

Do while loop are always at least executed once. even if the condition is evaluate to false.

```
let i = 0;
do{
    if(i%2!==0){
        console.log(i);
    i++;
    }
}while(i<=5);
// this code will be well executed.

// Now let put an error in the logic
let i = 9;
do{
    if(i%2!==0){
        console.log(i);
    i++;
    }
}while(i<=5);
// the console will return 9. becuz it's an odd number. The do will be execute. but the whi

// If you used a while loop:
let i=9;
while(i<=5){
    if(i%2!==0){
        console.log(i);
    i++;
    }
}
// The condition isn't correct now. The code will not be executed.
```

Infinite loops

You must avoid them. It can crash the browser and your pc.

Here's how you can create them:

```
let i = 0;
```

```
while(i<0){
    console.log(i)
    // This will create an infinite loop
    //i++; we forget to increment
}
// you can have infinite loop with
// do while, for loops
```

- You can't write `while(true)`.
- don't forget ton increment or decrement.
- take good care of the condition you are using.

For In and For OF loops

we Use them to iterate inside of an object or an array.

```
const person = {
    name:"Lelex",
    age:27;
}

for(let k in person){
    console.log(k, person[k]);
}

// ARRAY
const per = ['b','c','d','e'];

for(i in per){
    console.log(i,per[i]);
}
// in console
[Log] 0 - "b" (main.js, line 4)
[Log] 1 - "c" (main.js, line 4)
[Log] 2 - "d" (main.js, line 4)
[Log] 3 - "e" (main.js, line 4)
```

The ideal way to iterate over an Array is with the for-of loop.

```
// For-In
const per = ['b','c','d','e'];

for(i in per){
    console.log(i,per[i]);
}
// in console
[Log] 0 - "b" (main.js, line 4)
[Log] 1 - "c" (main.js, line 4)
```

```
[Log] 2 - "d" (main.js, line 4)
[Log] 3 - "e" (main.js, line 4)
```

```
// For-Of
const per = ['b','c','d','e'];

for(i of per){
    console.log(i);
}
[Log] b (main.js, line 4)
[Log] c (main.js, line 4)
[Log] d (main.js, line 4)
[Log] e (main.js, line 4)
```

best practice: YOu should use a for-in loop to iterate over element inside an object and a for-of loop to iterate over element inside an array.

Break and continue

```
//BREAK
let i = 0;
while(i<=10){
    if(i===5){
        break;
    }
    console.log(i);
    i++;
}
// In console
[Log] 0 (main.js, line 6)
[Log] 1 (main.js, line 6)
[Log] 2 (main.js, line 6)
[Log] 3 (main.js, line 6)
[Log] 4 (main.js, line 6)

//CONTINUE
let i = 0;
while(i<=10){
    // if(i===5){
    //     break;
    // }
    if(i%2 ===0){
        i++;
        continue;
    }
```

```
    console.log(i);
    i++;
}
//In console
[Log] 1 (main.js, line 10)
[Log] 3 (main.js, line 10)
[Log] 5 (main.js, line 10)
[Log] 7 (main.js, line 10)
[Log] 9 (main.js, line 10)
```

With the break keyword we jump out of a loop. With the continue keyword, we continue to another iteration.

Exercise - Chapter 4

1. function that return the max of two number.

```
function toMax(num1,num2){
    if(num1>num2){
        res = num1;
    }
    else{
        res = num2;
    }
    return res;
}
// in console
> toMax(123,500)
< 500
> toMax(600,500)
< 600

//OtherWay: conditional operator (ternary)
function toMax(num1,num2){
    return (num1>num2)? num1:num2;
}
```

2. Landscape or portrait.

```
// we will get an answer
function Islandscape(width,height){
    return (width>height) ? "Landscape":"Portrait";
}
// in console
> Islandscape(500,10)
< "Landscape"
> Islandscape(5,10)
< "Portrait"
```

```
// More easy and Pro
function Islandscape(width,height){
    return (width>height);
}
//in console
> Islandscape(5,10)
< false
> Islandscape(500,10)
< true
```

3. The Fizz Buzz algorithm

- Number divisible by 3 = fizz.
- Number divisible by 5 = Buzz.
- Number divisible by 3 and 5 = fizzbuzz.
- else return same number.
- catch non-number error: NOt a number.

```
function fizzbuzz(num){
    let answer;
    if(isNaN(num) || typeof(num)==='boolean'){
        throw Error("It's not a number");
    }
    else if(num%3 === 0 && num%5===0){
        answer = "fizzbuzz";
    }
    else if(num%3 === 0){
        answer = "fizz";
    }
    else if(num%5 === 0){
        answer = "buzz";
    }
    else{
        answer = num;
    }
    return answer;
}
// in console
> fizzbuzz(false)
< Error: It's not a number
> fizzbuzz(true)
< Error: It's not a number
> fizzbuzz('lelex')
< Error: It's not a number
> fizzbuzz(15)
< 15
```

```
> fizzbuzz(5)
< 5
> fizzbuzz(3)
< 3

// Mosh answer take.
// better way to do the if statement
if(typeof(num)!=='number')
// the code became

function fizzbuzz(num){
    let answer;
    if(typeof(num)!=='number'){
        throw Error("It's not a number");
    }
    else if(num%3 === 0 && num%5===0){
        answer = "fizzbuzz";
    }
    else if(num%3 === 0){
        answer = "fizz";
    }
    else if(num%5 === 0){
        answer = "buzz";
    }
    else{
        answer = num;
    }
    return answer;
}
```

4. Demerit points

- speed limits <= 70km -> ok
- 5km over the limit -> point: 1. Use Math.floor()
- [70,75[-> ok
- 80 -> point: 2
- more 12 points -> license suspended. (>130)

```
function checkspeed(speed){
    const speedlim = 70;
    const ppoint = 5;
    let ans;
    if(speed<75){  // speedlim+ppoint
        ans = "OK";
    }
    else{
        const point=Math.floor((speed-speedlim)/ppoint);
```

```
        if(point>=12){
            ans = "License suspended";
        }
        else{
            ans=console.log("Points:", point);
        }
    }
    return ans;
}
// in console
> checkspeed(72)
< "OK"
> checkspeed(715)
< "License suspended"
> checkspeed(126)
[Log] Points: - 11 (main.js, line 14)
```

5. Even and ODD Number.

```
function snum(limit){
    let ans;
    if(typeof(limit)!=='number'){
        throw Error('Please enter a number');
    }
    for(let i=0;i<limit;i++){
        if(i%2===0){
            ans=console.log(i+" Even Number");
        }
        else{
            ans=console.log(i+" ODD Number");
        }
    }
}
```

6. Count truthy

Create a function that count the truthy.

Remember the list of FALSY value in javascript:

- false
- ”
- null
- undefined
- NaN
- 0

```
function countruthy(arr){
    let n=0;
```

```
    for(let value of arr){
        if(value){ // if truthy
            n++ // the count will increment
        }
    }
    return  n;
}
```

7. Show strings properties in an object.

```
const amen = {
    name: "Valerie",
    money: 30000,
    function:"Business Woman"
}

function spro(obj){
    let ans;
    for(let key in obj){
        if(typeof(obj[key])==="string"){
            ans = console.log(key,obj[key]);
        }
    }
    return ans;
}

// In console
> spro(amen)
[Log] name - "Valerie" (main.js, line 16)
[Log] function - "Business Woman" (main.js, line 16)
```

8. Some of Multiple of 3 or 5

```
function mulsum(limit){
    let ans=0;
    for(let i = 0;i<=limit;i++){
        if(i%3===0||i%5===0){
            ans += i;
        }
    } // you must
    return ans; // return the ans out of the loop
}
// In console
> mulsum(10)
< 33
> mulsum(100)
< 2418
```

9. Calculate the average grade.

- return the grades of any given array
- 1-59: F
- 60-69:D
- 70-79: C
- 80-89: B
- 90-100: A

```
let a = [90,90,90,80]

function ravgrds(grades){
    // grades is an array
    let notes="";
    let sum = 0;
    let deno = grades.length;
    let avg = 0;
    for(let g=0;g<grades.length;g++){
        sum += grades[g];
        avg = sum / deno;
    }
    if(avg<60){
        notes = "F";
    }
    else if(avg<70){
        notes="D";
    }
    else if(avg<80){
        notes="C"
    }
    else if(avg<90){
        notes="B"
    }
    else{
        notes="A"
    }
    return console.log(notes,avg);
}

ravgrds(a);
// console
[Log] B - 87.5 (main.js, line 28)


// better way
// your function shouldn't handle to much things.
```

```
let a = [90,90,90,80]
// average calculation
function calavg(aw){
    let sum=0;
    for(let val of aw ){
        sum+=val;
    }
    return sum/aw.length;
}
// grade system
function calgrades(score){
    const avg = calavg(score);
    if(avg<60){
        notes = "F";
    }
    if(avg<70){
        notes="D";
    }
    if(avg<80){
        notes="C"
    }
    if(avg<90){
        notes="B"
    }
    else{
        notes =  "A";
    }
    return notes;
}
console.log(calgrades(a),calavg(a));
[Log] B - 87.5 (main.js, line 35)
```

For-of work well when we want the value not the index.

10. Show stars.

```
function showstars(rows){
    let ans;
    for(let r=1;r<=rows;r++){
        let pat="";
        for(let i =0;i<r;i++){
            pat+="*";
        }
        ans = console.log(pat);
    }
    return ans;
}
```

```
showstars(10);
// console
[Log] * (main.js, line 8)
[Log] ** (main.js, line 8)
[Log] *** (main.js, line 8)
[Log] **** (main.js, line 8)
[Log] ***** (main.js, line 8)
[Log] ****** (main.js, line 8)
[Log] ******* (main.js, line 8)
[Log] ******** (main.js, line 8)
[Log] ********* (main.js, line 8)
[Log] ********** (main.js, line 8)
```

11. Prime Numbers.

- divide by 1 and themselves

```
function showprime(limit){
    let ans;
    // 2 is first prime number
    for(let n=2;n<=limit;n++){
        let isprime = true;
        // 2 to current_number[n]
        for(let f=2;f<n;f++){
            if(n%f===0){
                isprime = false;
                break
            }
        }
        if(isprime){
            ans = console.log(n);
        }
    }
    return ans;
}

showprime(10);
// console
[Log] 2 (main.js, line 14)
[Log] 3 (main.js, line 14)
[Log] 5 (main.js, line 14)
[Log] 7 (main.js, line 14)

// better way

function showprime(limit){
```

```
    let ans;
    // 2 is first prime number
    for(let n=2;n<=limit;n++){
        if(isprime(n)){
            ans = console.log(n);
        }
    }
    return ans;
}

function isprime(n){
    // 2 to current_number[n]
    for(let f=2;f<n;f++){
        if(n%f===0){
            return false;
        }
    }
    return true;
}


showprime(10);
// console
[Log] 2 (main.js, line 6)
[Log] 3 (main.js, line 6)
[Log] 5 (main.js, line 6)
[Log] 7 (main.js, line 6)
```

Chapter 3 : Objects

> For all the litterature for this chapter, we must reference the javascript OOP documentation

basics

Put related variables inside an object

```
const circle = {
    radius: 1,
    location: {
        x:1,
        y:2
    },
    isVisible:true,
    draw: function(){
        console.log('draw');
    }
```

```
};
// dot notation
// bracket notion

circle.draw();
```

Object Oriented Programming style of programming.

Factory function and constructor function

```
// Factory function
function createCercle(radius){
    return {
        // radius:radius,
        // for simplicity
        radius,
        // for simplicity you can do
        // draw() {console.log('draw')}
        draw: function(){
        console.log('draw');
        }
    };
}
const circle1 = createCercle(1);
```

Type of notation convention

- Camel Notation: `abcDefgHijk`
- Pascal Notation: `AbcDefgHijk`

Working with constructor function, you should work with camel notation.

```
// constructor function
function Cercle(radius){
    this.radius = radius,
    this.draw = function(){
        console.log('draw');
    }
}
const circle2 = new Cercle(1);

// the new operator will create an empty object. The function will point the properties and
```

Dynamic Nature of programming

Even after the object has been created, you can add or remove properties and methods.

```
function Cercle(radius){
    this.radius = radius,
    this.draw =  function(){
        console.log('draw');
    }
}
const circle2 = new Cercle(1);

// let's add property  and method
circle2.color = 'red';
circle2.fon = function(){};

console.log(circle2);

//in Console
[Log] Cercle {radius: 1, draw: function, color: "red", fon: function} (main.js, line 12)

// Let's remove property and method
delete circle2.radius;
delete circle2.fon;

console.log(circle2);

// in console
[Log] Cercle {draw: function, color: "red"} (main.js, line 17)
```

Constructor Property

THe constructor property reference the function that was used to construct or create an object.

```
function Cercle(radius){
    this.radius = radius,
    this.draw =  function(){
        console.log('draw');
    }
}
const circle2 = new Cercle(1);

console.log(circle2);


// In console
> circle2.constructor
< function Cercle(radius){
    this.radius = radius,
```

```
    this.draw =  function(){
        console.log('draw');
    }
}

// Cercle our constructor function
// Cercle has his own constructor
// Function() is the built-in COnstructor
> Cercle.constructor
< function Function() {
    [native code]
}
```

Function are Objects

```
function Cercle(radius){
    this.radius = radius,
    this.draw =  function(){
        console.log('draw');
    }
}
const circle2 = new Cercle(1);

// the Cercle function has built-in method or property
console.log(Cercle.name);
```

Use built-in construct

```
// We can use the built-in function construct to
// build an object.
const Circle = new Function('radius',
`this.radius = radius,
    this.draw =  function(){
        console.log('draw');
    }`
);

const cerkle = new Circle(1);

console.log(cerkle);
```

Value types and reference types

Check OOP JS

Enumerating Properties of an objects

YOu can'y use a for-of loop on an object. Objects aren't iterable.

```
function Cercle(radius){
    this.radius = radius,
    this.draw =  function(){
        console.log('draw');
    }
}

const another = new Cercle(2);

// Enumerating properties and methods of an object
console.log(Object.keys(another));

for(let v in another){
    console.log(v,another[v]);
}

// this for-of will work though
for(let k of Object.keys(another)){
    console.log(k);
}

for(let k of Object.entries(another)){
    console.log(k);
}

// check

if('radius' in another){console.log('True')};

// in console
[Log] ["radius", "draw"] (2) (main.js, line 11)
[Log] radius - 2 (main.js, line 14)
[Log] draw - function (){ (main.js, line 14)
        console.log('draw');
    }
[Log] radius (main.js, line 19)
[Log] draw (main.js, line 19)
[Log] ["radius", 2] (2) (main.js, line 23)
[Log] ["draw", function] (2) (main.js, line 23)
[Log] True (main.js, line 28)
```

Cloning an Object

Copy the keys into an empty objects

```
function Cercle(radius){
    this.radius = radius,
    this.draw =  function(){
        console.log('draw');
    }
}

const another = new Cercle(2);



const copyC = {};

for(let k in another){
    copyC[k] = another[k];
}

console.log(another);
console.log(copyC);
// in console
[Log] Cercle {radius: 2, draw: function} (main.js, line 18)
[Log] {radius: 2, draw: function} (main.js, line 19)
```

Modern way to do it

```
function Cercle(radius){
    this.radius = radius,
    this.draw =  function(){
        console.log('draw');
    }
}

const another = new Cercle(2);


const copyC = {};

Object.assign(copyC,another);

console.log(another);
console.log(copyC);
// in javascript
[Log] Cercle {radius: 2, draw: function} (main.js, line 15)
[Log] {radius: 2, draw: function} (main.js, line 16)
```

```
function Cercle(radius){
    this.radius = radius,
```

```
    this.draw =  function(){
        console.log('draw');
    }
}

const another = new Cercle(2);


const copyC = {};
Object.assign(copyC,another);

const copcopy = Object.assign({color:"yellow"},another);

const cop = { ...another };

console.log(another);
// the copies
console.log(copyC);
console.log(copcopy);
console.log(cop);
// In console
[Log] Cercle {radius: 2, draw: function} (main.js, line 18)

[Log] {radius: 2, draw: function} (main.js, line 20)
[Log] {color: "yellow", radius: 2, draw: function} (main.js, line 21)
[Log] {radius: 2, draw: function} (main.js, line 22)
```

Garbage Collection

In C or C++ when we create an object, we need to allocate and deallocate memory to it. In Javascript by the time, we create that object the memory will be allocated. Because javascript has a garbage collector.

> A garbage collector will look for the variable and the constant that are no longer used and deallocate their memory.

Math Object

For mathematical calculation

> Math object in Javascript

String

```
// string primitive
const message = 'zobop';
// string object
const newMessage = new String('objecta');
```

```
// In console
> typeof message
< "string"
> typeof newMessage
< "object"

> message.length
< 5
> message.includes('o')
< true
> message[4]
< "p"
> message.startsWith('z')
< true
> message.endsWith('z')
< false
> message.indexOf('z')
< 0
> message.replace('p','t')
< "zobot"
// the message vairiable hasn't changed
> message
< "zobop"
```

However when we use the dot notation on a string primitive, internally the javascript engine wraps it with a string objects.

String object in javascript

- look for the escape notation

Template Literals

The template literals are indicated by the back-tick caractere.

```
const message = 'zobop \n is a zo';

const nmess = `Zobop
is a new zo`;

// in console
> nmess
< "Zobop
is a new zo"
> message
< "zobop
 is a zo"
```

It's very useful if you want to send email in your application. No need for escape notation.

You can had string dynamically with a place holder with the template literals.

```
let name = 'Jhon';

const mess = `${name} ${13+9}
is the right name`;

console.log(mess);

// in console
[Log] Jhon 22 (main.js, line 6)
is the right name
```

With the `${}` we can put any expression of function that produce a value.

Template Literals

Date

Date objects

```
const now = new Date();

// how to put string
const now1 = new Date(`May 11 2018 09:00`);

// the month start to 0 and finish to 11
const now2 = new Date(2018,4,11,9,0);

console.log(now);
console.log(now1);
console.log(now2);

// In console
[Log] Wed Mar 06 2019 14:09:51 GMT-0500 (EST) (main.js, line 9)
[Log] Fri May 11 2018 09:00:00 GMT-0400 (EDT) (main.js, line 10)
[Log] Fri May 11 2018 09:00:00 GMT-0400 (EDT) (main.js, line 11)

// format commonly use to put on web app
> now.toISOString()
< "2019-03-06T19:30:51.977Z"
```

Chapter 5 - Exercise

1. Adress Object

```
// street
// city
// zipcode
// showaddress

let address = {street:1, city:"P-au-P",zipcode:"ht6141"}


function showaddress(ad){
    for(let k in ad){
        console.log(k,ad[k]);
    }
}

showaddress(address);

// in console
[Log] street - 1 (main.js, line 11)
[Log] city - "P-au-P" (main.js, line 11)
[Log] zipcode - "ht6141" (main.js, line 11)
```

2. Factory and Constructor

```
// showaddress

// object literal
let address = {street:1, city:"P-au-P",zipcode:"ht6141"}
// factory
function addres(s,c,z){
    return{
        s,
        c,
        z
    };
}
const loc =  addres(2,"Jacmel","56045");
// constructor
function Addre(res,loca,zip){
    this.res = res;
    this.loca = loca;
    this.zip = zip;
}
const lok = new Addre(3,'Cap-Haitien','0009');

// show addresss
function showAddress(ad){
    for(let k in ad){
```

```
        console.log(k,ad[k]);
    }
}

showAddress(address);
showAddress(loc);
showAddress(lok);

// In console
[Log] street - 1 (main.js, line 25)
[Log] city - "P-au-P" (main.js, line 25)
[Log] zipcode - "ht6141" (main.js, line 25)
[Log] s - 2 (main.js, line 25)
[Log] c - "Jacmel" (main.js, line 25)
[Log] z - "56045" (main.js, line 25)
[Log] res - 3 (main.js, line 25)
[Log] loca - "Cap-Haitien" (main.js, line 25)
[Log] zip - "0009" (main.js, line 25)
```

Constructor function should have pascal notation.

3. Object Equality

2 objects are equals.

```
// object literal
let address = {street:1, city:"P-au-P",zipcode:"ht6141"}
// factory
function addres(s,c,z){
    return{
        s,
        c,
        z
    };
}
const loc =  addres(2,"Jacmel","56045");
// constructor
function Addre(res,loca,zip){
    this.res = res;
    this.loca = loca;
    this.zip = zip;
}
const lok = new Addre(3,'Cap-Haitien','0009');
const lok1  = new Addre(3,'Cap-Haitien','0009');

// Equal Objects
function eqObjects(ad1,ad2){
```

```
    let ans;
    for(let i in ad1){
        for(let j in ad2){
            if(ad1[i] === ad2[j]){
                ans = true;
            }
            else{
                ans = false;
            }
        }
    }
    return console.log(ans);
}

eqObjects(loc,lok);
eqObjects(loc,loc);
eqObjects(lok,lok1);

// in console
[Log] false (main.js, line 36)
[Log] true (main.js, line 36)
[Log] true (main.js, line 36)
```

But 2 objects are the same is different.

```
/ constructor
function Addre(res,loca,zip){
    this.res = res,
    this.loca = loca,
    this.zip = zip
}
const lok = new Addre(3,'Cap-Haitien','0009');
const lok1  = new Addre(3,'Cap-Haitien','0009');

// Are the same
function aresame(a1,a2){
    return a1 === a2;
}

// in console
> aresame(lok,lok)
< true
> aresame(lok1,lok)
< false
```

Two objects can be equals but they are two different objects in mem-

ory.

4. Blog post object

```
let blogPost = {
    title: `my blog`,
    body: `body blog`,
    author: `Alex`,
    views: function viewS(log){
        return log;
    },
    comments: [{
        author:`A viewer`,
        body: `Empty comment`
    },{
        author:`A viewer`,
        body: `Empty comment`
    }],
    islive: true
};
```

A scenario for building a blogging engine

```
// constructor function

function BlogPost(titre,auteur,body){
    this.titre = titre;
    this.auteur = auteur;
    this.body = body;
    // initialize
    this.views = 0;
    this.comments=[];
    this.islive = false;
}

let  post = new BlogPost(`Post`,`Lelex`,`Javascript is
a good programming language...`);

console.log(post);

// in console
[Log] BlogPost {titre: "Post", auteur: "Lelex", body: "Javascript is a good programming lang
> post.auteur
< "Lelex"
> post.body
< "Javascript is
a good programming language..."
```

5. Price range object

```
let priceRange = [{
    label:'$',
    tooltip:'inexpensive'
    minprice: 5,
    maxprice: 49
},{
    label:'$$',
    tooltip:'expensive'
    minprice: 50,
    maxprice: 150
}],{
    label:'$$$',
    tooltip:'Very-expensive'
    minprice: 151,
    maxprice: 500
};
```

We had these 4 properties for filtering possibilities.

Chapter 6 - Arrays

- adding new elements
- finding elements
- removing elements
- splitting arrays
- combining arrays

Adding new elements

```
const n = [3,4];

// end
n.push(5,6);
console.log(n);
// beginning
n.unshift(1,2);
console.log(n);
// middle
// the splice method is a bit tricky
// 3 args: start position, numbers of element
// to delete and then element to be added.
n.splice(2,0,`Array basics`);
console.log(n);

// in console
```

```
[Log] [3, 4, 5, 6] (4) (main.js, line 5)
[Log] [1, 2, 3, 4, 5, 6] (6) (main.js, line 8)
[Log] [1, 2, "Array basics", 3, 4, 5, 6] (7) (main.js, line 14)
```

const keyword doesn't stop us from modifying the content of an array.

Finding Elements

```
const n = [3,4];
const N = [3,3,3,5,6,77,8,66,77];

// indexof
// return -1 if false
i = n.indexOf(6);
console.log(i);

j = n.indexOf(4);
console.log(j);

// latest index of
k = N.lastIndexOf(3);
l = N.lastIndexOf(77);

console.log(k);
console.log(l);

// How includes work
console.log(n.indexOf(1)!==-1);
// better way
console.log(n.includes(1));

// indexof has an interesting second args
// the 6 means that the search will start at the
// 6 index
console.log(N.indexOf(77,6));

// in console
[Log] -1 (main.js, line 7)
[Log] 1 (main.js, line 10)
[Log] 2 (main.js, line 16)
[Log] 8 (main.js, line 17)
[Log] false (main.js, line 20)
[Log] false (main.js, line 22)
[Log] 8 (main.js, line 27)
```

finding reference types

```
const courses = [{id:1,name:"a"},{id:2,name:"b"}]
// The includes methods doesn't work


// find method
// 2 way for using callback function
// arrow function
const course = courses.find((c)=>{
    return c.name === 'a';
})

const course1 = courses.find(function(c){
    return c.id === 2;
})

// find index will return the index

const course2 = courses.findIndex((c)=>{
    return c.name === 'a';
})

const course3 = courses.findIndex(function(c){
    return c.id === 2;
})

console.log(course);
console.log(course1);
console.log(course2);
console.log(course3);
// in console
> course.includes({id:1,name:"a"})
< false

[Log] {id: 1, name: "a"} (main.js, line 26)
[Log] {id: 2, name: "b"} (main.js, line 27)
[Log] 0 (main.js, line 28)
[Log] 1 (main.js, line 29)
```

Finding Elements

A function used as an argument of another function is called a predicat or callback function.

`c=>c.name==="a"` is the ultimate shorthand in this case.

Removing Elements

```
const n = [1,2,3,4,5];
const N = [1,2,3,45,65,657,78,78,798];
console.log(n);
// remove from the End
let a = n.pop();
console.log(a);
// remove from the beginning
let b = n.shift();
console.log(b);
// remove from the middle
// start index 2 remove 3 elements
console.log(N.splice(2,3));
// in console
[Log] [1, 2, 3, 4, 5] (5) (main.js, line 3)
[Log] 5 (main.js, line 6)
[Log] 1 (main.js, line 9)
[Log] [3, 45, 65] (3) (main.js, line 12)
```

Emptying an array:

we can reassign the variable to an empty array but there is a catch

```
let n  = [1,2,3];
let o = n;

// reassignment
n = [];

// in console
> n
< [] (0)
> o
< [1, 2, 3] (3)
```

The garbage collector would normally erase the first array `[1,2,3]`. But the fact that we have another variable pointing to that object, it still will be in memory.

Les solutions:

```
let n  = [1,2,3];
let o = n;

// pick one
// solution 2
n.length = 0;
```

```
// solution 3
n.slice(0,n.length);

// solution 4
while(n.length>0){
    n.pop();
}
// console
> o
< [] (0)
> n
< [] (0)
```

Combining and Slicing Arrays

```
const f  = [1,2,3];
const n  = [4,5,6];

const com = f.concat(n);

// a copy of an array .slice()
// slice - default value is start index 0
// and number to be deleted or return 0
const slice = com.slice();

const slice1 = com.slice(1);
const slice2 = com.slice(0,3);
const slice3 = com.slice(0,4);


console.log(com);
console.log(slice);
console.log(slice1);
console.log(slice2);
console.log(slice3);

// in console
[Log] [1, 2, 3, 4, 5, 6] (6) (main.js, line 16)
[Log] [1, 2, 3, 4, 5, 6] (6) (main.js, line 17)
[Log] [2, 3, 4, 5, 6] (5) (main.js, line 18)
[Log] [1, 2, 3] (3) (main.js, line 19)
[Log] [1, 2, 3, 4] (4) (main.js, line 20)
```

An object is copied by his reference

```
const f  = [{id:1},2,3];
const n  = [4,5,6];
```

```
const com = f.concat(n);
const slice = com.slice();

console.log(com);
console.log(slice);

// a change
com[0].id = 100;


console.log(com);
console.log(slice);

// in console
[Log] [{id: 1}, 2, 3, 4, 5, 6] (6) (main.js, line 7)
[Log] [{id: 1}, 2, 3, 4, 5, 6] (6) (main.js, line 8)
[Log] [{id: 100}, 2, 3, 4, 5, 6] (6) (main.js, line 14)
[Log] [{id: 100}, 2, 3, 4, 5, 6] (6) (main.js, line 15)
```

the Spread operator

```
const f  = [{id:1},2,3];
const n  = [4,5,6];
// spread operator
const com = [...f, ...n];

const copy = [...com];

console.log(com);
console.log(copy);
// in console
[Log] [{id: 1}, 2, 3, 4, 5, 6] (6) (main.js, line 8)
[Log] [{id: 1}, 2, 3, 4, 5, 6] (6) (main.js, line 9)
```

Iterating an Array

Iterating in an array with a for-of loop.

```
const f  = [{id:1},2,3];
const n  = [4,5,6];
// spread operator
const com = [...f, ...n];

// for of
for(let k of com){
    console.log(k);
```

```
}

// for each
com.forEach((m)=>{
    console.log(m);
});
// you can display the index
com.forEach((m,index)=>{
    console.log(index,m);
});

// for in way can gives you index
// almost like the for each method

// in console
[Log] {id: 1} (main.js, line 8)
[Log] 2 (main.js, line 8)
[Log] 3 (main.js, line 8)
[Log] 4 (main.js, line 8)
[Log] 5 (main.js, line 8)
[Log] 6 (main.js, line 8)
[Log] {id: 1} (main.js, line 13)
[Log] 2 (main.js, line 13)
[Log] 3 (main.js, line 13)
[Log] 4 (main.js, line 13)
[Log] 5 (main.js, line 13)
[Log] 6 (main.js, line 13)
[Log] 0 - {id: 1} (main.js, line 17)
[Log] 1 - 2 (main.js, line 17)
[Log] 2 - 3 (main.js, line 17)
[Log] 3 - 4 (main.js, line 17)
[Log] 4 - 5 (main.js, line 17)
[Log] 5 - 6 (main.js, line 17)
```

Joining arrays

```
const f  = [{id:1},2,3];
const n  = [4,5,6];
// spread operator
const com = [...f, ...n];

console.log(com);

// const joined = com.join(',');
// the string arg is optional
const joined = com.join();
```

```
console.log(joined);
// in console
[Log] [{id: 1}, 2, 3, 4, 5, 6] (6) (main.js, line 6)
[Log] [object Object],2,3,4,5,6 (main.js, line 11)
```

Split is use in the case of strings

```
const mess = 'This is the work of the strings';
const parts = mess.split('');
// with spacing
const parts1 = mess.split(' ');

console.log(parts);
console.log(parts1);
//in console
// return an array
[Log] ["T", "h", "i", "s", " ", "i", "s", " ", "t", "h", …] (31) (main.js, line 18)
[Log] ["This", "is", "the", "work", "of", "the", "strings"] (7) (main.js, line 19)
```

```
const f  = [{id:1},2,3];
const n  = [4,5,6];
// spread operator
const com = [...f, ...n];

console.log(com);

// const joined = com.join(',');
// the string arg is optional
const joined = com.join('-');
console.log(joined);
// in console
[Log] [{id: 1}, 2, 3, 4, 5, 6] (6) (main.js, line 6)
[Log] [object Object]-2-3-4-5-6 (main.js, line 11)
```

Sorting arrays

```
const f  = [{id:1},2,3];
const n  = [4,5,6];
// spread operator
const com = [...f, ...n];
console.log(com);
// sort an array
// send the first at the last
com.sort()
console.log(com);
// revert an array
// change the complete order
```

```
com.reverse();
console.log(com);

//in console
[Log] [{id: 1}, 2, 3, 4, 5, 6] (6) (main.js, line 5)
[Log] [2, 3, 4, 5, 6, {id: 1}] (6) (main.js, line 9)
[Log] [{id: 1}, 6, 5, 4, 3, 2] (6) (main.js, line 13)
```

with Objects:

```
const courses = [{id:1,name:'Node'},{id:2,
    name:'Java'}];
console.log(courses);

courses.sort((a,b)=>{
    const A = a.name.toUpperCase();
    const B = b.name.toUpperCase();
    if(A<B){return -1;};
    if(A>B){return 1;};
    return 0;
});

console.log(courses);
// in console
[Log] [{id: 1, name: "Node"}, {id: 2, name: "Java"}] (2) (main.js, line 3)
[Log] [{id: 2, name: "Java"}, {id: 1, name: "Node"}] (2) (main.js, line 11)
```

ascii

In this code if we used a lowercase `'java'`, we won't see the change because of the position of uppercase N versus lowercase j on the ascii table.

Testing the elements of an Array

```
const n  = [-1,-2,-3,-4,-5,-6,-67,7];
const p = [5,-10];

// every() will check every element
// all the element are required to pass the test.
const atLeast = p.every(value=>{
    return value>=0;
});

// some()
// at least one element is supposed to past the
// test
const aOne = n.some(value=>{
    return value>=0;
```

```
});

console.log(atLeast);
console.log(aOne);

// in console
[Log] false (main.js, line 17)
[Log] true (main.js, line 18)
```

Filtering an array

```
const n  = [1,-2,3,-4,5,6,-67,7];

const f = n.filter(value=>{
    return value>=0;
});

console.log(f);

// in console
[Log] [1, 3, 5, 6, 7] (5) (main.js, line 7)
```

Mapping an Array

```
const n  = [1,-2,3,-4,5,6,-67,7];
const items = n.map(value=>{
    return '<li>'+value+'</li>';
});
const html = '<ul>'+items.join('')+'</ul>';
console.log(html);
// in console
[Log] <ul><li>1</li><li>-2</li><li>3</li><li>-4</li><li>5</li><li>6</li><li>-67</li><li>7</l
```

Map to objects

```
value => ({value});

const n  = [1,-2,3,-4,5,6,-67,7];
const items = n.map(value=>{
    return {value};
});

console.log(items);
// in console
[Log] [{value: 1}, {value: -2}, {value: 3}, {value: -4}, {value: 5}, {value: 6}, {value: -67
```

Chaining

```
const var = k.methdod().method2();
// or By convention
const var = k
    .methdod()
    .method2();
```

```
const n  = [1,-2,3,-4,5,6,-67,7];
const items = n
    .map(value=>{
        return {value};
    })
    .filter(obj =>{
        return obj.value>1;
    }
);

console.log(items);
// in console
[Log] [{value: 3}, {value: 5}, {value: 6}, {value: 7}] (4) (main.js, line 11)
```

We chained the mapping method with a filter method because the mapping outputed an array.

Reducing an array

```
const num  = [1,-2,3,-4,5,6,67,7];

let s = 0;
for(let n of num){
     s+=n;
}
console.log(s);

// second argument set to 0
// but it won't affect the actual calculation
// a= 0, b = 1 => a = 1
// a = 1, b = -2 => a = -1
// without the 0
// a = 1 the begining of the array directly
let ac = num.reduce((past,next)=>{
    return past+next;
},0);
console.log(ac);
// In console
[Log] 83 (main.js, line 7)
[Log] 83 (main.js, line 18)
```

Exercise - chapter 6

1. Array from Range

```
function arrRange(min,max){
    const arr = [];
    for(let k = min; k<=max;k++){
       arr.push(k);
    }
    return arr;
}

const numbers = arrRange(-1,10);
console.log(numbers);
// in console
> arrRange(-1,10)
< [-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, …] (12)
> arrRange(0,10)
< [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, …] (11)
> arrRange(0,100)
< [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, …] (101)
```

2. Make Own Includes

```
function incluD(array,tosearch){
    for(let i of array){
        if(i === tosearch){
            return true;
        }
        else{
            return false;
        }
    }
}

const testarr = [1,2,4,5,4,5,6,6,5,4,23];
const  test1 = incluD(testarr,22);
console.log(test1);

// in console
[Log] false (main.js, line 14)
```

3. Except

```
function except(array,torem){
    const output = [];
    for(let el of array){
        if(!torem.includes(el)){
```

```
            output.push(el);
        }
    }
    return output;
}

const testarr = [1,2,4,5,4,5,6,6,5,4,23];
const  test1 = except(testarr,[23,5,2]);
console.log(test1);

// in console
[Log] [1, 4, 4, 6, 6, 4] (6) (main.js, line 13)
```

4. Moving an Element

```
function move(array,index,offset){
    const position = index+offset;
    if(position>array.length || position<0){
        console.error('Invalid offset.');
        return;
        // the rest of the function won't be execute
    }
    const output = [...array];
    // delete
    const element = output.splice(index,1)[0];
    // addd
    output.splice(position,0,element);
    return output;
}

const testarr = [1,2,5,4,6,5,4,23];
console.log(testarr);
const  test1 = move(testarr,1,1);
console.log(test1);

// in console
[Log] [1, 2, 5, 4, 6, 5, 4, 23] (8) (main.js, line 15)
[Log] [1, 5, 2, 4, 6, 5, 4, 23] (8) (main.js, line 17)
```

5. Count Occurences

```
// basic
function count(array,search){
    let count =0;
    for(let k of array){
        if(k === search){
            count++
        }
```

```
    }
    return count;
}

const testarr = [1,2,5,4,6,5,4,23];
console.log(testarr);
const  test1 = count(testarr,4);
console.log(test1);

// inconsole
[Log] [1, 2, 5, 4, 6, 5, 4, 23] (8) (main.js, line 13)
[Log] 2 (main.js, line 15)

// clean
function count(array,search){
    return array.reduce((acc,current)=>{
        const occ = (current === search)? 1:0;
        return acc + occ;
    },0);
}


const testarr = [1,2,5,4,6,5,4,23];
console.log(testarr);
const  test1 = count(testarr,4);
console.log(test1);

// in console
[Log] [1, 2, 5, 4, 6, 5, 4, 23] (8) (main.js, line 18)
[Log] 2 (main.js, line 20)
```

6. Get max

```
// old school way
function getmax(array){
    if(array.length ===0){
        return undefined;
    }
    let ans = 0;
    for(let i = 0;i<array.length;i++){
        if(array[i]<array[i+1]){
            ans = array[i+1];
        }
        if(array[i]!==0){
            ans = array[i];
        }
    }
```

```
    return ans;
}



const testarr = [1,2,5,4,6,5,4,23,300,-100];
console.log(testarr);
const  test1 = getmax(testarr);
console.log(test1);

// in console
[Log] [1, 2, 5, 4, 6, 5, 4, 23, 300, -100] (10) (main.js, line 14)
[Log] 300 (main.js, line 16)
// case testarr = []
[Log] [] (0) (main.js, line 17)
[Log] undefined (main.js, line 19)

// mosh
function getmax(arr){
    if(arr.length === 0 ){
        return undefine
    }
    let max = arr[0];
    for(let i = 1;i<array.length;i++){
        if(array[i]>max){
            max = array[i+1];
        }
    }
    return max;
}
// clean
function getmax(array){
    if(array.length ===0){
        return undefined;
    }
   return array.reduce((a,c)=>(a>c)?a:c);
}
```

7. Movies

- all the movie in 2018 with rating > 4
- sort them by their rating
- descending order
- pick their title

```
const movies = [
    {titre:'a',year:2018,rating:4.5},
    {titre:'b',year:2018,rating:4.7},
```

```
    {titre:'c',year:2018,rating:3},
    {titre:'d',year:2017,rating:4.5}
];
console.log(movies);

const  film = movies
    .filter(m=>m.year===2018&&m.rating>=4)
    .sort((a,b)=>a.rating-b.rating)
    .reverse()
    .map(m=>m.titre);
console.log(film);
// in console
[Log] [{titre: "a", year: 2018, rating: 4.5}, {titre: "b", year: 2018, rating: 4.7}, {titre
[Log] ["b", "a"] (2) (main.js, line 14)
```

Chapter 7 - Functions

Function declaration vs expressions

In javascript we have two ways to define a function.

- function declasration
- function expression (named or anonymous)

```
// function declaration
function walk(){
    console.log('walk');
}
walk();

// function named expression

// can be a const
// similar to R
let run = function action(){
    console.log('Flash mode');
}
run();

// function anonymous expression
let speed = function(){
    console.log('Usain Bolt is scared')
}
speed();

// remmeber function are object.
```

```
// in console
[Log] walk (main.js, line 3)
[Log] Flash mode (main.js, line 12)
[Log] Usain Bolt is scared (main.js, line 18)
```

Hoisting

> The difference between the function declaration and the function expression:
> We can call the function `walk` using the function declaration syntax before the definition of the function. But this isn't possible with the function expression syntax.

```
// function declaration
walk();
function walk(){
    console.log('walk');
}

// function named expression
run();
let run = function action(){
    console.log('Flash mode');
}

// function anonymous expression
speed();
let speed = function(){
    console.log('Usain Bolt is scared')
}

// in console (2 errors)
[Log] walk (main.js, line 4)
[Error] ReferenceError: Cannot access uninitialized variable.
    Global Code (main.js:8)
[Error] ReferenceError: Cannot access uninitialized variable.
    Global Code (main.js:8)
```

definition of hoisting: At runTime the javascript engine will move all the function declaration at the top. They will be executed (interporeted first).

> Hoisting

Arguments

```
function sum(a,b){
    console.log(arguments);
    return a+b;
```

```
}
// an empty array has a length of 0
// but it ain't quite empty you will get 0

// a and b by default are undefined
console.log(sum(1)); // 1 + undefined
console.log(sum()); // undefined + undefined
console.log(sum(1,2,3,4,4,5,5,6))
 // the function will do 1+2 by default although
 // the arguments objects will record the other args

// the arguments property is an object

console.log(typeof(sum.arguments));

// we can work with argument object
// to simplify this code

function addition(){
    let total = 0;
    for(let add of arguments){
        total+=add;
    }
    return total;
}

console.log(addition(1,2,3,4,4,5,5,6));
console.log(addition(1,2,3,4,5,6,200,300));

// in console
[Log] Arguments [1] (1) (main.js, line 2)
[Log] NaN (main.js, line 9)
[Log] Arguments [] (0) (main.js, line 2)
[Log] NaN (main.js, line 10)
[Log] Arguments [1, 2, 3, 4, 4, 5, 5, 6] (8) (main.js, line 2)
[Log] 3 (main.js, line 11)
[Log] object (main.js, line 17)
[Log] 30 (main.js, line 30)
[Log] 521 (main.js, line 31)
```

The rest operator

The rest operator will put all our arguments inside an array. The rest operator has the same syntax as the spread operator with different usecase.

The rest parameter should be the last parameter of a function.

```
function add(...args){
```

```
    console.log(typeof(args));
    console.log(args);
    // args is an array (array is an object)
    return args.reduce((a,b)=>a+b);
}
console.log(add(1,2,3,4,4,5,5,6));
console.log(add(1,2,3,4,5,6,200,300));

// in console
[Log] object (main.js, line 15)
[Log] [1, 2, 3, 4, 4, 5, 5, 6] (8) (main.js, line 16)
[Log] 30 (main.js, line 20)
[Log] object (main.js, line 15)
[Log] [1, 2, 3, 4, 5, 6, 200, 300] (8) (main.js, line 16)
[Log] 521 (main.js, line 21)
```

Practical example

```
/ rate is in the range of 0.1 to 0.9
function discount(rate,...prices){
    const total = prices.reduce((a,b)=>a+b);
    return total *(1-rate);
}
console.log(discount(0.6,50000,90909090,30495595905));
// in console
[Log] 12234621998 (main.js, line 28)
```

Default Parameters

```
function interest(principal,rate,years){
    // default value
    rate = rate||3.5;
    years = years||5;

    return ((principal*rate)/100)*years;
}
console.log(interest(10000,3.5,5));
console.log(interest(10000));
// in console
[Log] 1750 (main.js, line 8)
[Log] 1750 (main.js, line 9)

// default value (2mode)
function interest(principal,rate=3.5,years=5){
    return ((principal*rate)/100)*years;
}
console.log(interest(10000,3.5,5));
console.log(interest(10000));
```

```
// in console
[Log] 1750 (main.js, line 5)
[Log] 1750 (main.js, line 6)
```

when you give an operator a default value, `2mode` (nb: 2mode is my creation to avoid confusion), we must also give a default value to the folllowing parameters.

```
// default value
function interest(principal,rate=3.5,years){
    return ((principal*rate)/100)*years;
}
console.log(interest(10000,3.5,5));
console.log(interest(10000));
// console
[Log] 1750 (main.js, line 5)
[Log] NaN (main.js, line 6)

// to bypass
console.log(interest(10000,3.5,5));
console.log(interest(10000,undefined,5));
//console 2
[Log] 1750 (main.js, line 5)
[Log] 1750 (main.js, line 6)
// undefined will reference the rate parameter and
// the fact that the year parameter wasn't defined by default.
// it's bad practice
```

Getters and setters

At this point, we will how to set or exposed properties of an object for utilies.

```
// display a person fullname
const person= {
    fname:"Alexandro",
    lname:"Disla",
    // two way to play with function in object
    nom : function(){
        return `${person.fname} ${person.lname}`;
    },
    // no need of keyword function
    nomComplet(){
        return `${person.fname} ${person.lname}`;
    }
};
```

```
console.log(person.nom());
console.log(person.nomComplet());

// console
[Log] Alexandro Disla (main.js, line 15)
[Log] Alexandro Disla (main.js, line 16)
```

Getters will put it to read-only.

- getters allow us to access a property
- setter allow us to change or mutate a property

```
// display a person fullname
const person= {
    fname:"Alexandro",
    lname:"Disla",
    get nomComplet(){
        return `${person.fname} ${person.lname}`;
    },
    set entry(value){
        const parts= value.split(' ');
        this.fname = parts[0];
        this.lname = parts[1];
    }
};
console.log(person.nomComplet);

// lets set a new name
person.entry='Lelex AD0791'
console.log(person);

// in console
[Log] Alexandro Disla (main.js, line 14)
[Log] Object (main.js, line 18)
entry:
fname: "Lelex"
lname: "AD0791"
nomComplet:
Object Prototype
```

try and catch

Error handeling is also called defensive programming.

```
// display a person fullname
const person= {
    fname:"Alexandro",
    lname:"Disla",
```

```
    get nomComplet(){
        return `${person.fname} ${person.lname}`;
    },
    set entry(value){
        //defensive programming

        // this way the error will not pass
        // the code will not run
        // if(typeof(value)!=='string')return;

        // another way
        // throw  new error object
        // Error() is a constructor function
        // We throw an exception
        if(typeof(value)!=='string'){
            throw new Error("It's not an string");;
        }
        const parts= value.split(' ');
        if(parts.length!==2){
            throw new Error("Enter fname and lname");;
        }
        this.fname = parts[0];
        this.lname = parts[1];
    }
};
console.log(person.nomComplet);

// handle the exception
try{
person.entry=null;
}
catch(e){
    // e reference the error object
    console.log(e);
    alert(e);
};
try{
    person.entry='';
}
    catch(e){
        // e reference the error object
        console.log(e);
        alert(e);
    };
```

```
console.log(person);
// in console
[Log] Alexandro Disla (main.js, line 30)
[Log] Error: It's not an string - main.js:20 (main.js, line 38)
[Log] Error: Enter fname and lname - main.js:24 (main.js, line 46)
[Log] {fname: "Alexandro", lname: "Disla"} (main.js, line 51)
```

Keep in mind that `alert` is an older to handle exception.

Local and scope

```
const mess = "hi";
console.log(mess);
// in console
[Log] hi (main.js, line 2)
//
// but
// { here is a code block}
{
    const mess = "hi";
}
console.log(mess);
//in console
[Error] ReferenceError: Can't find variable: mess
    Global Code (main.js:6)
```

> A scope of a variable or constant determine where that variable or console is accessible. A global variable or constant is accessible everywhere. local variable and constant take precedence over global variable and constant.

```
// global constant
const color  = 'red'

function start(){
    const mess = "hi";
    const color = 'blue'
    console.log(color); // blue will be display
    // local var or const > globala var or constant
    // its bad practice though
    console.log(mess); // good
}
console.log(mess); // we will get error

if(true){
    const mes2 = "hi";
    console.log(mes2); // good
```

```
}
console.log(mes2); // we will get error

for(let i=0;i<5;i++){
    console.log(i); // good
}
console.log(i) // we will get error


function start(){
    const mess = "hi"; // valid
    console.log(mess); // good
}

function stop(){
    const mess = "hi"; // valid
    console.log(mess); // good
}
```

let and var

var comes with a lot of problems.

```
// let vs var

for(let i =0;i<10;i++){
    console.log(i);
}
console.log(i); // scope issue
// console
[Log] 0 (main.js, line 4)
[Log] 1 (main.js, line 4)
[Log] 2 (main.js, line 4)
[Log] 3 (main.js, line 4)
[Log] 4 (main.js, line 4)
[Log] 5 (main.js, line 4)
[Log] 6 (main.js, line 4)
[Log] 7 (main.js, line 4)
[Log] 8 (main.js, line 4)
[Log] 9 (main.js, line 4)
[Error] ReferenceError: Can't find variable: i
    Global Code (main.js:6)

// var case
for(var i =0;i<10;i++){
    console.log(i);
```

```
}
console.log(i); // scope issue disapear
// console 2
[Log] 0 (main.js, line 4)
[Log] 1 (main.js, line 4)
[Log] 2 (main.js, line 4)
[Log] 3 (main.js, line 4)
[Log] 4 (main.js, line 4)
[Log] 5 (main.js, line 4)
[Log] 6 (main.js, line 4)
[Log] 7 (main.js, line 4)
[Log] 8 (main.js, line 4)
[Log] 9 (main.js, line 4)
[Log] 10 (main.js, line 6)
```

with `var i =0;`, `i` is accessible out of the scope of the block of the for loop.

- ES6 (Ecmasscript2015): `let, const` create block-scoped variables
- `var` creates function-scoped variable

The `var` attach our variables to the window object. (which is bad practice)

```
// on global variable
var color = 'red';
let age = 27;
// console
> window
< Window {document: #document, NaN: NaN, color: "red", window: Window, Infinity: Infinity, ..
> window.color
< "red"
```

When we define a function, that function is a global function. That function is attach to the global object `window`.

The this keyword

> `this` references the object that is executing the current function.

- for a method, it references the object (function are object, object).
- for a function, it references the global object (window,global).

the call back function is just a regular function. Used in an object, it won't be considered as a method. The function are attach to the gobale object.

```
// call back is not a method
const video = {
    title:'xbox1',
    player:['aria','aegon','cali','lelex'],
    show(){
        this.player.forEach(function(t){
```

```
            console.log(this);
        });

    }
}
video.show();
// console
// the element of player are attach to the
// global object: window
[Log] Window {document: #document, NaN: NaN, window: Window, Infinity: Infinity, undefined:
[Log] Window {document: #document, NaN: NaN, window: Window, Infinity: Infinity, undefined:
[Log] Window {document: #document, NaN: NaN, window: Window, Infinity: Infinity, undefined:
[Log] Window {document: #document, NaN: NaN, window: Window, Infinity: Infinity, undefined:
//
// but
const video = {
    title:'xbox1',
    player:['aria','aegon','cali','lelex'],
    show(){
        this.player.forEach(function(t){
            console.log(this);
        },this);
        // this as second arguments assure us that
        // the call back function is operating as
        // a method
    }
}
video.show();
//console 2
[Log] {title: "xbox1", player: ["aria", "aegon", "cali", "lelex"], show: function} (main.js
[Log] {title: "xbox1", player: ["aria", "aegon", "cali", "lelex"], show: function} (main.js
[Log] {title: "xbox1", player: ["aria", "aegon", "cali", "lelex"], show: function} (main.js
[Log] {title: "xbox1", player: ["aria", "aegon", "cali", "lelex"], show: function} (main.js
// we are sure that the call back function
// is behaving as a method.
```

the answer now.

```
const video = {
    title:'xbox1',
    player:['aria','aegon','cali','lelex'],
    show(){
        this.player.forEach(function(t){
            console.log(this.title,t);
        },this);
        // this as second arguments assure us that
        // the call back function is operating as
```

```
        // a method
    }
}
video.show();
// console
[Log] xbox1 - "aria" (main.js, line 6)
[Log] xbox1 - "aegon" (main.js, line 6)
[Log] xbox1 - "cali" (main.js, line 6)
[Log] xbox1 - "lelex" (main.js, line 6)
```

Change this

```
// not quite a good practice
const video = {
    title:'xbox1',
    player:['aria','aegon','cali','lelex'],
    show(){
        const self = this;
        this.player.forEach(function(t){
            console.log(self.title,t);
        });
    }
}
// self is referencing the object
// the callback function will behave like a method.
video.show();
//console
[Log] xbox1 - "aria" (main.js, line 7)
[Log] xbox1 - "aegon" (main.js, line 7)
[Log] xbox1 - "cali" (main.js, line 7)
[Log] xbox1 - "lelex" (main.js, line 7)
```

Function in javascript are also object. We have 3 methods `apply, call, blind` that we can used to change the value of `this`.

```
function playvideo(q,b){
    console.log(this);
}
// this is referencing the window object
// now
playvideo.call({almamaterre:"SLG"},1,2);
playvideo.apply({almamaterre:"SLG"}),[1,2];
// bind creates a new function from playvideo function
playvideo.bind({almamaterre:"SLG"})();
playvideo();

// console
```

```
[Log] {almamaterre: "SLG"} (main.js, line 16)
[Log] {almamaterre: "SLG"} (main.js, line 16)
[Log] {almamaterre: "SLG"} (main.js, line 16)
[Log] Window {document: #document, NaN: NaN, playvideo: function, window: Window, Infinity:
```

Now

```
const video = {
    title:'xbox1',
    player:['aria','aegon','cali','lelex'],
    show(){
        this.player.forEach(function(t){
            console.log(this.title,t);
        }.bind(this));
    }
}
// with the bind method from the call back
// we are referencing the object
video.show();
// console
[Log] xbox1 - "aria" (main.js, line 6)
[Log] xbox1 - "aegon" (main.js, line 6)
[Log] xbox1 - "cali" (main.js, line 6)
[Log] xbox1 - "lelex" (main.js, line 6)
```

Arrow function can help

```
const video = {
    title:'xbox1',
    player:['aria','aegon','cali','lelex'],
    show(){
        this.player.forEach(t=>{
            console.log(this.title,t);
        });
    }
}
// => inherit the this from the
// containing function  and assure use that
// we are sure that the callback will behave well.
// thanks Es6
video.show();
// console
[Log] xbox1 - "aria" (main.js, line 6)
[Log] xbox1 - "aegon" (main.js, line 6)
[Log] xbox1 - "cali" (main.js, line 6)
[Log] xbox1 - "lelex" (main.js, line 6)
```

Arrow function saves life!

Exercise - CHapter 7

1. Sum of arguments

The rest operator convert the args, we pass to an array.

```
// numbers or arrays
function sum(...items){ // rest operator
    if(items.length===1 && Array.isArray(items[0])){
        // spread operator
        items = [...items[0]];
    }
    return items.reduce((a,b)=>a+b);
}
console.log(sum(1,2,3,4,5,65435,545,434,324,24,34));
console.log(sum([1,2,3,4,5,65435,545,434,324,24,34]));
// console
[Log] 66811 (main.js, line 9)
[Log] 66811 (main.js, line 10)
```

2. Area Circle

```
// radius  settable
// area read-only

const circle = {
    radius: 10,
    get area(){
        return Math.PI*this.radius*this.radius ;
    }
    // set radius(num){
    //     radius = num;
    // }
};

console.log(circle.area);
// console.log(circle.area);
//console
[Log] 314.1592653589793 (main.js, line 14)
```

3. Handeling error

```
try{
    const numbers  = [1,2,3,4,5,5,5566];
    const count = countOcc(null,5);
    console.log(count);
}
catch(e){
    console.log(e.message);
```

```
}
function countOcc(array,selem){
    if(!Array.isArray(array)){
        throw new Error('Bad Array son!');
    }
    return array.reduce((acc,curr)=>{
        const occ = (curr === selem)? 1:0;
        return occ+acc;
    },0);
}
// console
// the error is caught
```
