Javascript

-> Talk about Javascript:
- Used in browser to make web page interactive.

-> Due to intense community support and investment of big companies:
- web/ Mobile apps.
- real time network apps.
- Command line Tools.
- Games.

-> Javascript run:
- The browsers have a JavaScript Engine (firefox: SpiderMonkey; Chrome:v8)
- Node = v8 + c++

-> EcmaScript vs JavaScript:
- ecmascript: specification = reponsible for defining standards or features for javascript
- JavaScript: Programming Language

JavaScript for Beginners

Basics

We are working on the browser console.
-> hello world in javascript

```
console.log("hello world");
```

-> the basics operation still work:
- P.E.M.D.A.S + Modulo
- Logical operators

-> create an alerte with the alerte function

```
alerte('yo')
```

Yo will be display in the web browser.

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 deploy our code into the browser simultaneously.

Where to put the 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.

We are working in visual studio code
> 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>
```

Run the same code with node

with Node: Open the terminal and go to the specific directory.

```
$ pwd
/Users/alexandrodisla/Dropbox/JavaScript-Youtube
$ ls
index.html      index.js
$ node index.js
Hello World
```

Node is a runtime environment for executing js code.

Programming State

Variable

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 reassing the value of a variable. Use "const" when you a declaring and initiate the value of the variable.

In JS we have two categories of types :

- primitives/value types
- references types

type of programming Languages

- static (statically-typed)
  > Static typing means that types are known and checked for correctness before running your program. This is often done by the language'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)
  > 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: "'python 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

Primitive types

we have: 1. string 2. number 3. boolean 4. undefined 5. null

```
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!!!!!
```

In javascript we don't have floating point number.
"Selectedcolor" is an object.

References Types

We have: - 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

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 varaibles 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.
```

Object-Oriented Programming (OOP)

Before OOP we had uniquely procedural programming (functional programming). It's a straight forward style programming where you are only using functions on the variables (the data).
RIsk Of Spaghetti codes: To much interdependancy between the functions. That's where OOP solved the problem. In OOP, we group the related functions and variables into a unit called object. We refer to these variables as properties and the functions as methods.
Definition OOP is a programming paradigm (style of programming) centered around objects rather than functions. OOP is not a programming language or a tool.

-> Languages that support OOP:
- C/C++
- C#
- Java
- Ruby
- Python
- Javascript - R, PHP, ...

-> The (popular) frameworks are designed with OOP concepts (for example Angular).

4 Pillars of OOP

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism

Encapsulation

> Encapsulation is the fact that we regroup the related funcions and variables into a unit called Object. In this object, the variables are refered as properties and the the functions as methods.

```
// procedural programming
let baseSalary = 4600;
let overtime = 4;
let rate = 0;

function getwage(baseSalary,overtime,rate){
    return baseSalary + (overtime*rate);
}
// you would do
getwage();

// object oriented programming
// we create the object employeeMPCE
let employeeMPCE = {
    baseSalary : 4600,
    overtime : 4,
    rate : 0,
    getwage: function(){
        return console.log(this.baseSalary + (this.overtime*this.rate));
    }
```

```
};
employeeMPCE.getwage(); //method
console.log((employeeMPCE.baseSalary)); // property
// the fewer parameters a function has, the easier it is to maintain that function.
```

Abstraction

It's when you can hide some of the properties and methods from the outside. This gives us a couple of benefits:
- make the interface simpler **(an object with few properties and methods is easier then an object with several properties and methods)**
- reduce the impact of change

Inheritance

It's a mecanism that allow you to eliminate redundant code. Let's take an example from html. Sandbox, textbox and checkbox, these 3 elements have properties like hidden, innerHTML and methods like click and focus. Instead of redifining all the properties and methods for each of these html elements, we can **create a generic object call htmlelement, and have other objects inherit his properties and methods**

-> it looks like parent-child in css

Polymorphism (many forms)

Polymorphism is a technique that allows you to eliminate long if and else or switch and case statements. For example if you have 3 html objects that you want to render. With the procedural programming we would have to use long conditional statements (if and else or switch and case). But with OOP, we can define a render method for each of these objects. And each of them would behave differently depending on the type of these objects.

Benefits of OOP

- encapsulation = reduce complexity + increase reusability
- abstraction = reduce complexity + isolate impact of change
- inheritance = eliminate redundant code
- polymorphism = refactor ugly switch and case (conditional statements) statements

Objects in Javascript (<OOP>)

- creating objects (way 1)
- factories and constructors (way 2 3)
- primitives and reference types
- working with properties
- private properties
- getters and setters

Creating an object

An object in javascript is a collection of key value pairs.

```
// object Literal
let employeeMPCE = {
    baseSalary : 4600,
    overtime : 4,
    rate : 0,
    getwage: function(){
        return console.log(this.baseSalary + (this.overtime*this.rate));
    }
};
employeeMPCE.getwage();
console.log((employeeMPCE.baseSalary));
// or
let circle = {
    radius: 1,
    location:{
        x:1,
        y:2
    },
    draw:function(){
        console.log('draw');
    }
};
circle.draw();
// draw is a method
// location and radius are properties
// properties (variables) are used to hold values.
// A method (function) is use to define some logic.
```

Object literal is a simple way to define an object. But **we can also**

define an object with factories and constructors.

Factories and constructors

With object literal, if we want to create a new similar object you would have to duplicate the object. It's not a good practice to do so. If an object have more then one method we shouldn't duplicate them. Now when an object have more then one method, it is said that this object has a behavior. The object literal way is limited.

Duplicating an object is a problem if the object has behavior.

Factory function (create objects)

To solve the problem

```
// factory function
function dcircle(radius,x,y){
    return {
        radius,
        x,
        y,
        draw:function(){
            console.log("factory");
        }
    };
}
// Now in Javascript, when creating a factory function,
// the properties can be use as an parameters

const circle = dcircle(1,1,1); // we create (by use) the object

circle.draw(); // access the function (since the function is log)
console.log(circle.radius); // access the value
```

Constructors function (another way to create object)

The naming convention of an constructors is different. We must start with an uppercase letter at the beginning. The constructor method looks like your are creating an instance of a classe. In Javascript we don't have the concept of classes.

```
// factory function
function dcircle(radius,x,y){
    return {
        radius,
        x,
        y,
```

```
        draw:function(){
            console.log("factory");
        }
    };
}
// Now in Javascript, when creating a factory function,
// the properties can be use as an parameters

const circle = dcircle(1,1,1); // we create (by use) the object

circle.draw(); // access the function (since the function is log)
console.log(circle.radius); // access the value
//////////////////////

//constructors

function Cercle(r,u,t){
   // console.log("this", this) // reference the global object
    this.r = r; // this : the reference executing the code
    this.u = u; // this : indicate an empty object in memory
    this.t = t; // This : set the properties and methods
    this.trace = function(){
        console.log("Constructors");
    }
}
const autre  =  new Cercle(2,2,2);
autre.trace();
console.log(autre.u);
// the new operator
// will create under the hood an empty object. It will set
// "this." to point to that object.
// if we remove the new, it will return the window object
// depending of the browser. bad practice.
// global object in browser is call window object.
// global object in node is call globa object.
```

Constructor property

> Every object in js has a property called constructor and that references the function that was used to create that object. When we create an object with the object literal syntax internally the javascript engine uses this constructor function. If we do this in the browser.

```
> Cercle.constructor // log this
< function Function() {
    [native code]
}
```

```
// That Function() is the constructor.
////////////
new String(); // vs the string literal : "", ''
new Boolean(); // vs the boolean literal : true, false
new Number(); // vs the Number literal : 1, 2, 3, ...
// it's cleaner (good practice) to use those literals than their object equivalant.
```

Functions are objects in Javascript

```
function Cercle(r,u,t){
    this.r = r;
    this.u = u;
    this.t = t;
    this.trace = function(){
        console.log("Constructors");
    }
}
const autre  =  new Cercle(2,2,2);
autre.trace();
console.log(autre.u);
// the Cercle function has method and properties
// in the console of our browser we will see it better
> Cercle.constructor
< function Function() {
    [native code]
}
// proof that a function is a constructor
// function is an object in javascript
const autre1 =  new Function('r','u','t',`
this.r = r;
this.u = u;
this.t = t;
this.trace = function(){
    console.log("Constructors");
}
`)
const slik = new autre1(1,1,1);
// another tricks
 // autre.call({},1,1,1)
// is exactly like doing new autre1(1,1,1)
// call is the method of  thi function autre1
// autre.apply({},[1,1,1])
// is the same as slik and slik1
// apply is another method of the function
// `` is use to make multiple line strings
// we use the built in Function constructor to create
```

```
// the function autre1
// in console
> slik
< anonymous {r: 1, u: 1, t: 1, trace: function}
```

primitives and reference types

Primitives or value types :
- Number
- String
- Boolean
- Symbol
- undefined
- null

reference types:
- object
- function
- array

We see that array and function are technically objects

Primitives

-> Work with primitives

```
let x = 100;
let y = x; // copy
x=200;
// in the console
> x
< 200
> y
< 100
// the value of y is independent form the new assignment of x.
// Other case let's change x to an object literal
let x ={value:100};
let y = x; // copy is different
x.value = 200;
// Now the value of y will also change.
// in the console
> x
< {value: 200}
> y
< {value: 200}
```

When we use an object. That object is not stored in this variable. That object is stored somewhere else in the memory. And the address of that memory location is stored inside that variable. So then when

we copy x into y, it's the address or the reference that is copied. In other word both x and y are pointing the same object in memory. Then when we modify that object using x or y. It changes our immidiatly visible to the other variable.

- Primitives are copied by their value.
- Objects are copied by their reference.

```
let n  = 10;
function inc(n){
    n++; // increment -> n = n+1
    return console.log(n);
}
inc(n); // 11
console.log(n); // 10
// in to the console the ouput is 11 and the other 10.
// the variable defined outside the function is
// independent from the parameter of that function
// increment will not make effect on the n define //out off the scope of the function.
////////////////////////////////////////////
// Now let's create an object literal
let o  = {value:10};
function inc(o){
    o.value++; // increment -> n = n+1
    return console.log(o);
}
inc(o);
console.log(o);
// in the console
[Log] {value: 11} (index.js, line 4)
[Log] {value: 11} (index.js, line 7)
```

Adding and removing Properties

Objects in javascript are dynamic. That's mean after you create them, you can add or delete properties.

-> add property

```
function Cercle(r,u,t){
    this.r = r;
    this.u = u;
    this.t = t;
    this.trace = function(){
        console.log("Constructors");
    }
}
```

```
const autre  =  new Cercle(2,2,2);
autre.trace();
console.log(autre.u);
 // add properties in Javascript is easy
 // we don't have classes in JS
 autre.diam = {m:4};
 // in the console
> autre
< Cercle {r: 2, u: 2, t: 2, trace: function, diam: {m: 4}}
```

-> you can use the bracket notation instead of the the dot notation. Also in the case that the property name isn't a valid identifier.

Valid Identifiers

```
 "centre location"
// centre location is not a valid identifier. Same as Centre-location.
// you can't use the dot notation on either one of them.
autre["centre location" ]
```

-> Delete property

```
// delete a property
delete autre.diam; // or with bracket notation
//In the console
> autre
< Cercle {r: 2, u: 2, t: 2, trace: function}
// the property diam is delete.
```

> When you are dynamicly adding or removing a property of an object.
> It is good practice to it with the bracket notation.

```
const pname = "centre location";
// autre[pname] = {key:value};
```

Enumerating properties

```
function Cercle(r,u,t){
    this.r = r;
    this.u = u;
    this.t = t;
    this.trace = function(){
        console.log("Constructors");
    }
}
const autre  =  new Cercle(2,2,2);
autre.trace();
console.log(autre.u);
```

```
for(let key in autre){
    console.log(key, autre[key])
}
// output the keys and their values
/////////
// return the methods only:
for(let key in autre){
    if(typeof(autre[key]) !== 'function'){
        console.log(key, autre[key])
    }
}
 //another method is to use directly
//the Object built-in function-object.
console.log(Object.keys(autre));
//check is a key is in an object.
if("r" in autre){
    console.log(true)
}
```

Abstraction

Hide the details and show the essentials.

```
// everything is accessible
function Cercle(radius){
    this.radius = radius;
    this.dloc = {x:0, y:0};
    this.comoploc = function(){
        // ...
    };
    this.trace = function(){
        this.comoploc();
        console.log("Constructors");
    };
}
const autre  =  new Cercle(2);
autre.trace();
console.log(autre.radius);
```

-> let's learn how to do it

Private properties and methodes

this.a = a set a as property. but you can define it as a local variable to hide it in the constructor function.

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

```
    let dloc = {x:0, y:0}; // private property
    this.trace = function(){
        console.log("Constructors");
    };
    Object.defineProperty(this, "dloc",{
        get: function(){
            return dloc;
        }, //
        set: function(value){
            if(!value.x || !value.y){
                throw new Error("invalid location");
            }
            dloc = value;
        }
    });
}
const autre  =  new Cercle(2);
autre.trace();
console.log(autre.radius);
console.log(autre.dloc); // we are accessing this private
// private property with this getter
autre.dloc = 1; // we access and set the value externally
// with the setters
```

The End