JS Weird Parts 30 - 50

Udemy:
https://www.udemy.com/understand-javascript/learn/v4/overview

Repository:
C:\Users\Lenovo\Desktop\Isti\Programozás-\Kurzusok\Javascript\Udemy-Understanding-weird-Parts

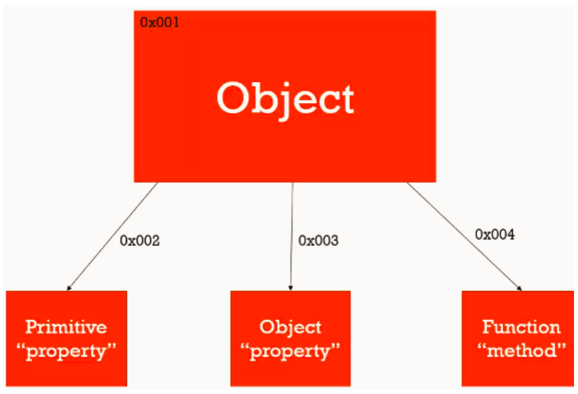
Github:
Useful Link:
http://underscorejs.org/
https://lodash.com/

Section 4

Lesson_30 Objects and Functions

They are really related!

Objects are collection of named value pairs.
So all the object and all of it's properties are allocated to the memory and the main object is just referencing this standalone stuff stored seperated in memory.



Lesson 31 Object Literals

JavaScript Object Literal:
A JavaScript object literal is a comma-separated list of name-value pairs wrapped in curly braces. Object literals encapsulate data, enclosing it in a tidy package. This minimizes the use of global variables which can cause problems when combining code.
The following demonstrates an example object literal:

```
var myObject = {
    sProp: 'some string value',
    numProp: 2,
    bProp: false
};
```

You can create objects on the fly.

```
var Tony = {
    firstname: 'Tony',
    lastname: 'Alicea',
    address: {
        street: '111 Main St.',
        city: 'New York',
        state: 'NY'
    }
};

function greet(person) {
    console.log('Hi ' + person.firstname);
}


greet(Tony);
//Created on the fly
greet({
    firstname: 'Mary',
    lastname: 'Doe'
});
```

Lesson 32 Faking NameSpaces

Namespace: is typically a container for functions and variables

```
 //The second greet will overwrite the first greet variable
var greet = 'Hello!';
var greet = 'Hola!';


console.log(greet);
// THIS the same as the following


// english.greetings = "Hello":
var english = {
    greetings: {
        basic: 'Hello!'
    }
};


var spanish = {};
```

```
spanish.greet = 'Hola!';


console.log(english);

console.log(spanish.greet);
```

Lesson 34 JSON not Object Literals

JSON = Java**S**cript**O**bject**N**otation

Differencies to JSON:

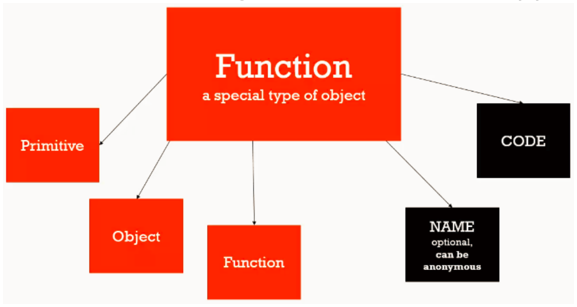
"properties has to wrapped in quotes"

```
//Creating a simple objct literal

var objectLiteral = {

    firstname: 'Mary',

    isAProgrammer: true

}


//converting object literal to json

console.log(JSON.stringify(objectLiteral));

//Transfer from JSON to object literal

var jsonValue = JSON.parse('{ "firstname": "Mary", "isAProgrammer": true }');


console.log(jsonValue);


// JSON

// {

//     "firstname": 'Mary',

//     "isAProgrammer": true

// }
```

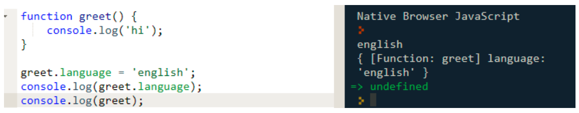
Lesson 34 Functions are objects

First Class functions: everything you can do with the other types, you can do them with the functions.

Function is an object, which "code" happened to be one of the property of the object.



You can attach a property to a function, since that is an object as well.



```
function greet() {
    console.log('hi');
}

greet.language = 'english';
console.log(greet.language);
console.log(greet);
```

```
Native Browser JavaScript

english
{ [Function: greet] language:
'english' }
=> undefined
```

Lesson 35 Function Statements and Expressions

Expression:
A unit of code which result in a value

```
a = 3
```

Statement:
A unit of code which is only executed

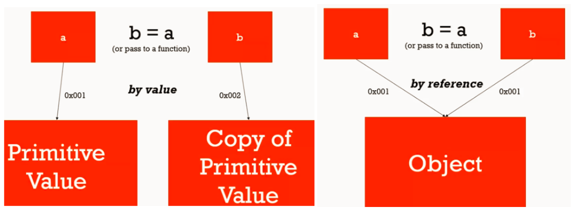
Only **functions**, and **variables** are **hoisted**, stored in the memory.

```
// Function statement, it will create a function object, stores it's properties
function greet() {
    console.log('hi');
}

greet.language = 'english';
console.log(greet.language);
//anonymusGreet(); this would turn out "undefined"
//Function expression, the variable name, is poiting into th stored funciton object
//First it will create a variable, and will store as "undefined", then on the fly we
create the ojbect
```

```
var anonymusGreet = function(){
    console.log('hi');
};
anonymusGreet();
```

Lesson 36 By Value, By Reference

mutate: To change something



Objects assigned to each other, they are just pointing to the same object, they have the same reference to the same object

```
var c = { greeting: 'hi' };
var d;
d = c;
```

```
// by value (primitives)
var a = 3;
var b;

b = a;          //It will create a new spot in memory for b, and it will copy a
a = 2;
//Copies of each other, they are stored in two different spots in memory
console.log(a);
console.log(b);

// by reference (all objects (including functions)) are stored as reference
```

```
var c = { greeting: 'hi' };
var d;

d = c;
c.greeting = 'hello'; // mutate

console.log(c);
console.log(d);

// by reference (even as parameters)
function changeGreeting(obj) {
    obj.greeting = 'Hola'; // mutate
}

changeGreeting(d);
console.log(c);
console.log(d);

// equals operator sets up new memory space (new address)
c = { greeting: 'howdy' };
console.log(c);
console.log(d);
```

Lesson 37 `This`

- function statement, function express will refer to the global object
- Object property function called, will refer to the object

```
//function statement
function a() {
    console.log(this);
    this.newvariable = 'hello';   //This will be "undefined"
}
//function expression
var b = function() {
    console.log(this);            //This will be "undefined"
}

a();

console.log(newvariable); // not good!

b();

var c = {
    name: 'The c object',
    log: function() {
        var self = this;                //this referring to the object, in which it is
called

        self.name = 'Updated c object';
        console.log(self);

        var setname = function(newname) {
            self.name = newname;                //since we have stored the object
reference with "this", we will remember the object as "this"
            this.name = newname;                //Since we are in a function
expression, the global variable will be ponted with "this"
        }
        setname('Updated again! The c object');
        console.log(self);
    }
}

c.log();
```

Lesson 39 - arguments

The arguments variable, means, it will represent the function objects's argument array properties.

```
function greet(firstname, lastname, language) {

    console.log(arguments);

}
```

Lesson 40 Function OverLoading

A **functions, can't overWrite each other**'s. like in Java

```
//Shared functions, instead of overloading them

function greet(firstname, lastname, language) {


    language = language || 'en';


    if (language === 'en') {

        console.log('Hello ' + firstname + ' ' + lastname);

    }


    if (language === 'es') {

        console.log('Hola ' + firstname + ' ' + lastname);

    }


}


function greetEnglish(firstname, lastname) {

    greet(firstname, lastname, 'en');

}
```

```
function greetSpanish(firstname, lastname) {
    greet(firstname, lastname, 'es');
}


greetEnglish('John', 'Doe');
greetSpanish('John', 'Doe');
```

Lesson 42 Automatic Semicolon Insertion (return)

1. Put always the semicolon on the end of the line!
2. In case of `return` especially

```
function getPerson() {


//This doesn't work since the js engine automatically puts a semicolon here
    return
    {
        firstname: 'Tony'
    }
//This will work, since js knows that we starte an object literal
    return {
        firstname: 'Tony'
    }
}
console.log(getPerson());
```

Lesson 43 WhiteSpace

Invisible charachters that create literal "space" in your written code.
Very Liberal

```
 var
```

```
    // first name of the person

    firstname,


    // last name of the person

    lastname,


    // the language

    // can be 'en' or 'es'

    language;


var person = {

    // the first name

    firstname: 'John',


    // the last name

    // (always required)

    lastname: 'Doe'

}


console.log(person);
```

Lesson 44 Immediatley invoked function expressions

You can immediately, call the function, with the arguments

```
// IIFE

(function(){

})();

// function statement

function greet(name) {

    console.log('Hello ' + name);

}
```

```
greet('John');


// using a function expression
var greetFunc = function(name) {
    console.log('Hello ' + name);
};
greetFunc('John');


// using an Immediately Invoked Function Expression (IIFE)
var greeting = function(name) {

    return 'Hello ' + name;

}('John');

console.log(greeting);

// IIFE
var firstname = 'John';

(function(name) {

    var greeting = 'Inside IIFE: Hello';
    console.log(greeting + ' ' + name);

}(firstname)); // IIFE
```

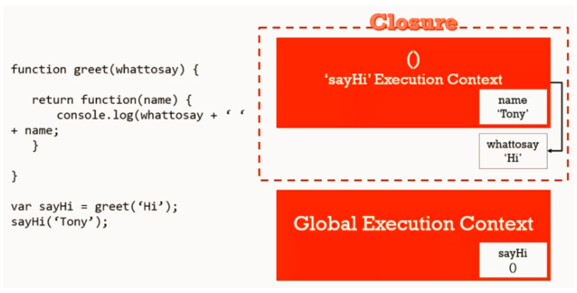
Lesson 45 IIFE

```


// IIFE crreats it's own execution context
(function(global, name) {
    var greeting = 'Hello';

    global.greeting = 'Hello';

    console.log(greeting + ' ' + name);


}(window, 'John')); // IIFE


console.log(greeting);
```

Lesson 47 Closure Part1-2 (ES6)

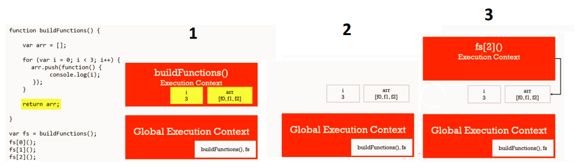
When the function executional context finishes, the variables put into the memory are still there. (Until the garbge collection comes)

Closure, a function will include and close all the variables to which they should have access.



```
function greet(whattosay) {

   return function(name) {
      console.log(whattosay + ' '
+ name;
   }

}

var sayHi = greet('Hi');
sayHi('Tony');
```

1. Function is creating it's own execution context, i which it is allocation two variables into the memory,
2. After it has allocated the variables to the memory, until the garbage collection is not coming it will stay there
3. When the created functions is called, it will know where to reference the (previously created and yet not deleted variables)



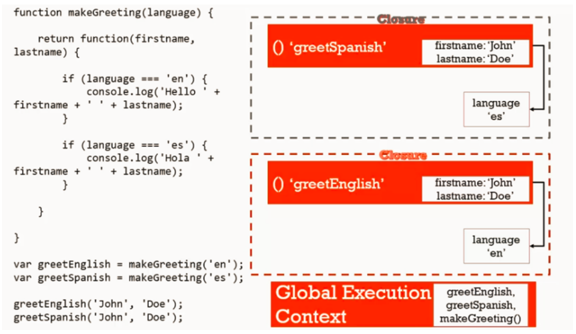
```
function buildFunctions() {

    var arr = [];

    for (var i = 0; i < 3; i++) {
      arr.push(function() {
           console.log(i);
      });
    }

    return arr;

}

var fs = buildFunctions();
fs[0]();
fs[1]();
fs[2]();
```

```
// Making an array to have a collection of functions
```

```
//When the function is returned, we are storing the variable of i= 3 and the functions
pushed to the array
function buildFunctions() {


    var arr = [];
    for (var i = 0; i < 3; i++) {
        arr.push(
            function() {
                console.log(i);
            }
        )
    }
    return arr;
}
//Store the array in the memory and call the functions
var fs = buildFunctions();
//Here the functions are invoked
fs[0]();            //3
fs[1]();            //3
fs[2]();            //3


//with ES6, local enviromental variable, using the "let" keyword
//with this we are assigning the value of the i variable to a different variable, for a
different scope, stored seperately in the memory
//thus a different j will be stored for the functions in the array
// function buildFunctions2() {
//     var arr = [];
//     for (var i = 0; i < 3; i++) {
//         let j = i;
//         arr.push(
//             function() {
//                   console.log(j);
//             }
//         )
//     }
//     return arr;
// }

//In order to create a seperately executable context, to create different scopes for
the variable
//we use immediately invokable function, which has it's own excetuinal context, with
different stored variables
function buildFunctions2() {
    var arr = [];
    for (var i = 0; i < 3; i++)  {
        let j = i;                          //with ES6, local enviromental variable
        arr.push(
            (function(j) {
                return function() {
                    console.log(j);
                }
            }(i))
        )
    }
    return arr;
}

var fs2 = buildFunctions2();

fs2[0]();
fs2[1]();
fs2[2]();
```

Lesson 48 Function Factories

So where the function get executed and in that executonal context if an other function gets created, it will remember the created variables and when it will need them it will just close them in it's own exceutional context and use them.



```
function makeGreeting(language) {

    return function(firstname,
lastname) {

        if (language === 'en') {
            console.log('Hello ' +
firstname + ' ' + lastname);
        }

        if (language === 'es') {
            console.log('Hola ' +
firstname + ' ' + lastname);
        }

    }

}

var greetEnglish = makeGreeting('en');
var greetSpanish = makeGreeting('es');

greetEnglish('John', 'Doe');
greetSpanish('John', 'Doe');
```

```
// Event after the  makeGreeting function execution context is finished, i can still
access to the "language" variable

//So we can simply create functions, which determines the language at once, and when we
need it we just have to call them with the addintional info,

function makeGreeting(language) {


    return function(firstname, lastname) {


        if (language === 'en') {

            console.log('Hello ' + firstname + ' ' + lastname);

        }



        if (language === 'es') {

            console.log('Hola ' + firstname + ' ' + lastname);

        }



    }



}



var greetEnglish = makeGreeting('en');
var greetSpanish = makeGreeting('es');


greetEnglish('John', 'Doe');

greetSpanish('John', 'Doe');


/*Prevoies Function OVerloading function, where we created create 1 function, whith all
of the parameters*/

//Shared functions, instead of overloading them

// function greet(firstname, lastname, language) {


//     language = language || 'en';


//     if (language === 'en') {

//         console.log('Hello ' + firstname + ' ' + lastname);

//     }


//     if (language === 'es') {

//         console.log('Hola ' + firstname + ' ' + lastname);

//     }


// }


// function greetEnglish(firstname, lastname) {

//     greet(firstname, lastname, 'en');

// }


// function greetSpanish(firstname, lastname) {

//     greet(firstname, lastname, 'es');

// }


// greetEnglish('John', 'Doe');

// greetSpanish('John', 'Doe');
```

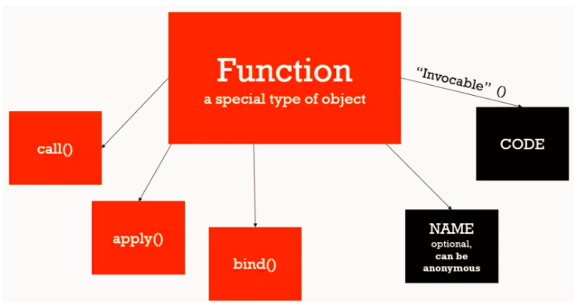
49 Closures and Callbacks

Closures are used all the time, for example at **setTimeout**().
Callbacks : is a function executed when other function is finished.
First-class function: which can be passed around as arguments

```
// 1. sayHiLater function finishes running, the variable of greeting is still in the memorypace, in the execution context

// 2. then it will execute the setTimeout function, after 3 seconds, then it will check if any functions is listening to it's executions

// 3. Callback is a function executed when other function is finished.


function sayHiLater() {

    var greeting = 'Hi!';

 //This is taking a function object as a parameter

    setTimeout(function() {

        console.log(greeting);

    }, 3000);

}

sayHiLater();
```

50 Call, Apply, Bind

Every function objects has these three properties.



bind() It will **make a copy of the function object** and inside this newly created function "this" will refer to the passed object in the bind.

call(): Pass coma seperated arguments
Apply(): Pass an array of arguments
They are **executing the function**, with the passed object and arguments

```
//Created object to reference.

var person = {

    firstname: 'John',

    lastname: 'Doe',

    getFullName: function() {

//since it will be executed inside an object, "this" will refer to the object

        var fullname = this.firstname + ' ' + this.lastname;

        return fullname;


    }

}


//here right now "this" refers to the window object, unless we use apply, or bind
functions

//since this function "getFullName" is not part of the window object

var logName = function(lang1, lang2) {


    console.log('Logged: ' + this.getFullName());

    console.log('Arguments: ' + lang1 + ' ' + lang2);

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


}


//Here we are using the built of bind() property of any function object, where we pass
an object, which will be referred later as "this"

var logPersonName = logName.bind(person);

//Called later, with tha already passed varible

logPersonName('en');
```

```
//It is executing the function, with the passed object and arguments

logName.call(person, 'en', 'es');          //Pass coma seperated arguments

logName.apply(person, ['en', 'es']);        //Pass an array of arguments
```

// apply, can be used in IIFE as well!

```
(function(lang1, lang2) {


    console.log('Logged: ' + this.getFullName());

    console.log('Arguments: ' + lang1 + ' ' + lang2);

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


}).apply(person, ['es', 'en']);
```

Function Borrowing

```
var person2 = {

    firstname: 'Jane',

    lastname: 'Doe'

}


console.log(person.getFullName.apply(person2));


// function currying

function multiply(a, b) {

    return a*b;

}


var multipleByTwo = multiply.bind(this, 2);

console.log(multipleByTwo(4));


var multipleByThree = multiply.bind(this, 3);

console.log(multipleByThree(4));
```

50 Functional programming

SImilar to other languages as "schema".
Due to passable first-class functions we can **think and code in terms of functions.**
Resuable Components!

//Functonal Programming, we are segmeting our code in a clean way, relying on these first-class functions
//We put work into functions.
//It will take an array, it will loop through the array, and it will execute the passed function on the current loop

1. We are creating a function, which takes a function as an argument, which well be called over the looping

```
function mapForEach(arr, fn) {

    var newArr = [];

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

        newArr.push(

            fn(arr[i])

        )

    };

    return newArr;

}

var arr1 = [1,2,3];

console.log(arr1);

var arr2 = mapForEach(arr1, function(item) {

   return item * 2;

});

console.log(arr2);

var arr3 = mapForEach(arr1, function(item) {

   return item > 2;         //will return a boolean.

});

console.log(arr3);
```

2. we are creating a function, which will preset, the function argument, in a way we want

```
//an easy limier function

var checkPastLimit = function(limiter, item) {

    return item > limiter;

}

//We are passing the "checkPastLimit()" function as a functional argument

// We are using bind in order to make a copy of the above function, "this" will refer
to anything, doesn't matter

// 1 will refer to the limiter, and when it will be caled in the mapForEach, the item
will be the current loop

var arr4 = mapForEach(arr1, checkPastLimit.bind(this, 1));

console.log(arr4);
```

3. **We are wrapping the function in a limiter function, presetting to the created function the limiter with bind and the result will be a returned function which can be inserted into the mapForEach function argument**

```
var checkPastLimitSimplified = function(limiter) {

    return function(limiter, item) {

        return item > limiter;

    }.bind(this, limiter);

};


var arr5 = mapForEach(arr1, checkPastLimitSimplified(1));

console.log(arr5);
```

51 UnderScore library for efficient code

http://underscorejs.org/
https://lodash.com/