Javascript Foundations (pdf)

> **Son Güncellenme Tarihi:** 03.10.2023
> **©Copyright:** Onur Dayibasi

1. Variables

Why did we change from javascript var variable to let,const ?

- no var → Global Scope
- var → Function Scope
- let/const → Block Scope

1.1 What is the purpose of the var variable?

var is an abbreviation for variable. These variables are used to store names, messages, numbers, and dates and to perform actions on them in the software.

```
var name="Onur";  var height=180;  var age;
```

So, what if we never utilized it as we described below? To begin with, age cannot be defined alone. JS must recognize it as an assignment in order to function. For the others (name, height), JS will create and assign a variable called name in the global scope.

```
name="Onur";  height=180;  age;
```

When it comes to using functions, developers don't want their function values to be accessed from outside. They just want it processed at that level of function. Because the var variable is not used on the left, we see a foo variable that can be used outside of the function in the example below.

The distinction between using "var"

But what we always want as developers is for the variables we specify to live, be seen, and be accessed within the scope we define. Outside of this, accesses become unpredictable as the amount of code develops and it is uncertain where the variable is changed from.

Another key criterion is that variables specified in the upper scope can be accessed and utilized by variables defined in the lower scope. This is already a favorable scenario.

```
var foo='World';
function sayHello(){
    console.log(`Hello ${foo}`);
    for(var i=0;i<2;i++) {
        console.log(`Again ${foo}`)
    }
}
sayHello();
```

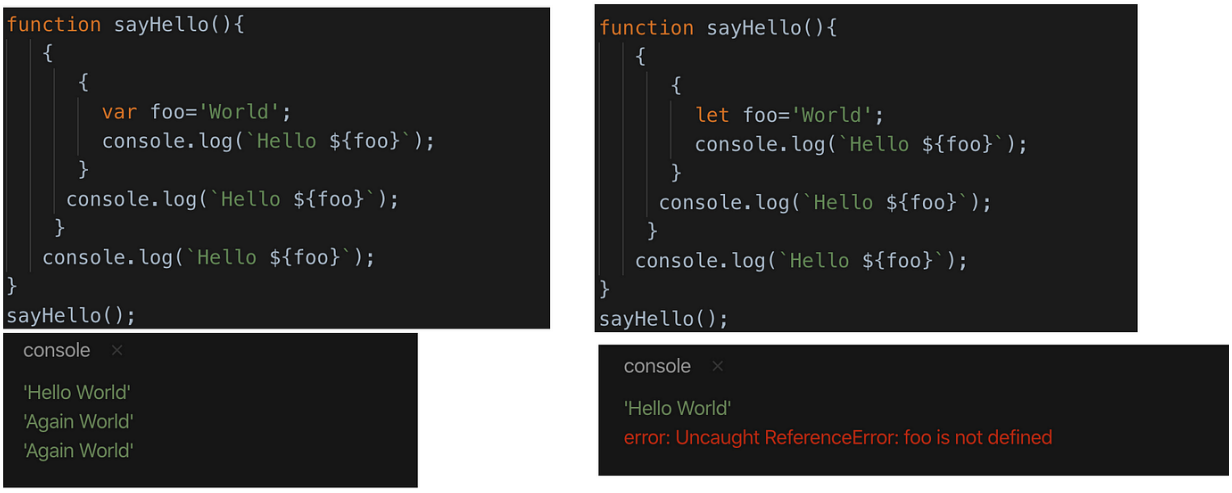
```
console
'Hello World'
'Again World'
'Again World'
```

variable defined in file scoped

1.2 What is the purpose of the let/const variable?

We were satisfied with the function scope at this point. It was exclusively used in scope definitions like for/while, if/switch, and functions. However, when our requirement to create code in the form of nested blocks with async/promise, arrow function, and callback grew, var in the function became insufficient.

Although the var variable on the left side defines the innermost scope, it may also be utilized in the top scope. Instead, we want the inner scope variable to be inaccessible from the outside, as seen on the right side.



Nested scope definition

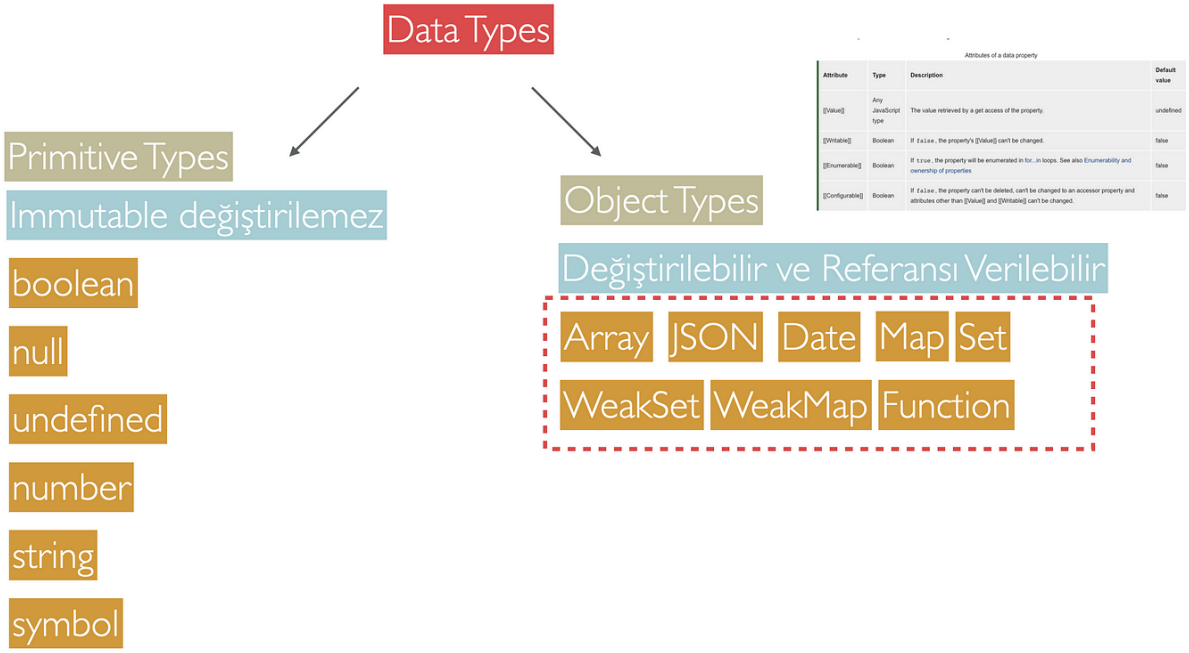
After the initial assignment, const stops you from making any further assignments to the variable. If you define a PI constant and then try to alter it, JS will produce an error.

1.3 Example Code

- Samples

2. Data Types

To understand the mechanics of how JavaScript works, we must first understand the JavaScript Engine and how it handles various data types. This section will concentrate on the Data Types used in JS.



JavaScript Data Types

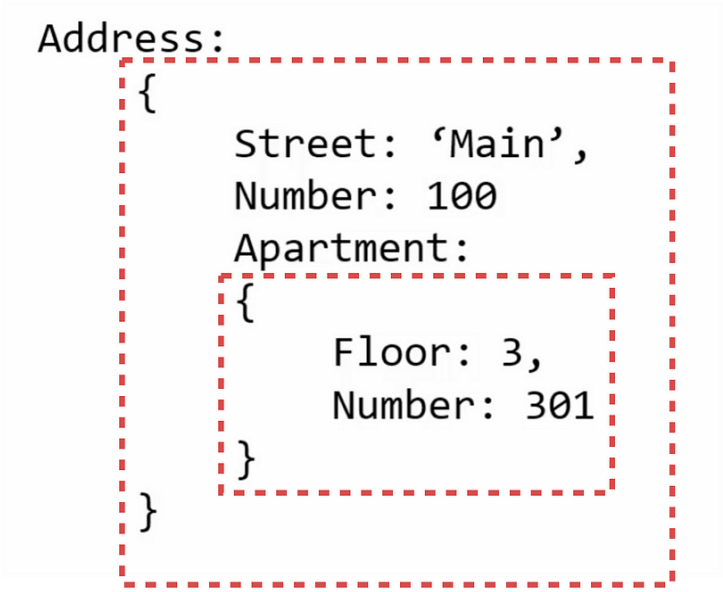
JavaScript Data Types are separated into two categories, as seen in the diagram above. The first is Primitive Type, while the second is Object Type. In the ADT (Abstract Data Types) structure, I go into length about this. But, to summarize once more:

2.1 Primitive Characteristics

When a variable is assigned a JS Primitive value (boolean, integer, null, undefined, string, symbol), it takes up memory and its value is not modified. These are known as immutable data types.

2.2 Types of Objects

If a variable is specified as an Object type, the values in the memory space that this variable stores can be altered, and an Object can include Primitive and other Object Types several times. These are known as mutable data types because their contents can be altered. For instance, (Object, Array, Function, Date, and so on.)



Creating a Nested Value Object Structure

3. Operators

What exactly do we mean by JS Operator? Do these have a higher priority? What is the JS Engine's approach to these operators? In this post, I will go over the operators *, /, +, and >,, =, >=,===, ==,!=, &&, || in greater detail.

Different visuals are used to invoke operator specified functions.

3.1 Types of Operators

3.1.1 Numerical Operators

, /, +, - are special characters for multiplication, division, addition, and subtraction.

For instance, consider the + operator.

```
const result = 1+2 //It performs the operation of adding two integers.r
console.log(result) //3
```

Well, let's imagine that we have a sum method in the background.

```
function sum(a,b){ return ...//}
const result= sum(1,2);
console.log(result) //3
```

In this case, it is like we are calling **+** and **sum** function. We could have written the code as follows.

```
function +(a,b){ return ...//}
const result= +(1,2);
console.log(result) //3
```

Here the + at the beginning is the prefix calling property. JS executes the + operation in the middle, this is called infix, and if we specify it at the end, this is called postfix.

```
+(1,2)  //prefix
(1+2)   //infix
(1,2)+  //postfix
```

Below you can see examples of Infix, Prefix, Postfix calls. In this case we can say that they are actually specialized functions for the operator.

Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

Infix, Prefix, Postfix

Similarly, **multiply, subtract, divide** operators are similarly specialized functions. In addition to this

Reminder **%** returns the **remainder** of the value divided by **%**.

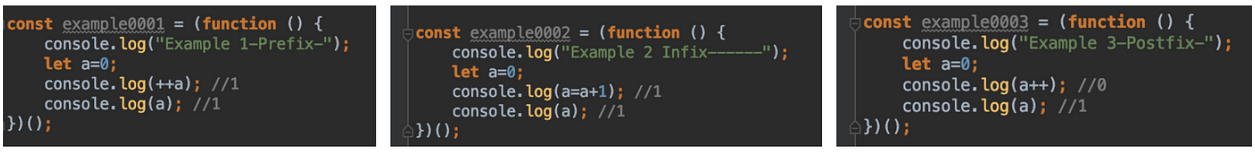
```
console.log(7%3) //1
```

Some of these functions are new ES6 features. ** is a short form of exponentiation. For example

```
console.log(2**4) //16
```

3.1.2. Increment/Decrement Operators

Already Increment ++ and Decrement - Operator can be used with all kinds of approaches. Here, since the postfix operation is incremented after returning, you can see this change when you move to the bottom line.



Increment / Decrement sample

3.1.3. Comparison Operators

We looked at mathematical operators above, but if we use > as below, we compare the value on the 2 sides of the operator and the result is not numeric but bool(true/false)

```
console.log(1>2) //false
```

> is large, < is small, >=large is equal, <=small is equal, == equal, !== not equal, === type and value equal, !== type and value not equall,

```
console.log("1>2",1 >2); //false
console.log("1<2", 1 < 2); //true
console.log("1<=2", 1 <= 2); //true
console.log("1>=2", 1 >= 2); //false
console.log("1!=2", 1 != 2); //true
console.log(`"1" !== 1`, "1" !== 1); //true
console.log(`"1" != 1`, "1" != 1); //false
console.log(`"1" === 1`,"1" === 1); //false
console.log(`1 === 1`,1 === 1); //true
```

in allows you to check if there is a prop in the object, instanceof allows you to check if this variable is of the corresponding type

```
const obj = {x: 10, y: 20};
console.log("x" in obj); //true
console.log("z" in obj); //false
console.log(obj instanceof Object) //true
```

3.1.4. Logical and Bitwise Operators

Circuits with logical 2 bool values interact with one another to generate outcomes. Gate Here or Gate or | &&& And Gate

```
console.log(true && false); //false
console.log(true && true); //true
console.log(false || true); //true
```

If we want to perform bitwise operations on integer values by creating them in a system of 2, then we can use the & AND, | OR or ^ XOR operators.

```
const a = 5;        // 00000000000000000000000000000101
const b = 3;        //00000000000000000000000000000011
```

```
console.log(a & b); // 00000000000000000000000000000001
// expected output: 1
```

3.1.5. Assignment Operators

The most common assignment method is the = expression. It assigns the value on the right side to the left side.

```
let x=3;
```

You can make operations while making these assignments. There are many examples like this *= Assign the value by multiplying. **= Assign the value by calculating the value. /= Assign by dividing the value, etc…

3.1.6 Bitwise Shift Operators

You can decrease the value of the number >> to the right or increase it << to the left.

```
const a = 2;        // 000000000000010
console.log(2 << a); //8
console.log(a >> 1);// 1
```

3.1.7 Conditional Operators

condition ? ifTrue : ifFalse After a certain conditional evaluation, if the condition is valid after the ? sign, it is executed, otherwise it is executed after :

.? does not go forward if the value obj is null..

```
const obj = {
    position: {x: 10}
}

console.log(obj.position.x) //10;
console.log(obj.pos.x) //Exception
console.log(obj.pos?.x) //undefined
```

?? returns the value to its right if the checked value is null/undefined.

```
const foo = null ?? ‘default string’;
console.log(foo);
```

3.1.8 Functional Programming Operators

arrow function allows to define functions to be used to define pure function types in functional programming.

```
sum =(a,b) => a+b
```

The **pipe operator** again provides a structure that makes the function chain more readable in the functional programming approach.

```
/* This example from
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Pipeline_opera
tor
**/
const double = (n) => n * 2;
const increment = (n) => n + 1;

// without pipeline operator
const result0=double(increment(double(double(5)))); // 42
console.log(result0)

// with pipeline operator
const result1= 5|> double|> double|> increment|> double; // 42
console.log(result1)
```

3.1.9 Unary Operators

These operators take a single parameter and deduce a result from this value.

```
delete Student.name //prop deletes
typeof xxx //returns the type of the variable
void
Adds by converting +Number if the first variable is numeric.
- Subtracts by converting Number. if the first variable is numeric.
~ Bitwise NOT operator.
! Takes a note (converts boolean
!!! Note Takes note (converts boolean)
```

3.2 Operator Precedence

What are these operator priorities? As you can see in the picture below, Operator priorities are ordered, for example Grouping is at the front.

```
console.log(3*3-2) //7;
console.log(3*(3-2)) //3 The group is executed first.
```

Precedence	Operator type	Associativity	Individual operators
21	Grouping	n/a	`( … )`
20	Member Access	left-to-right	`… . …`
	Computed Member Access	left-to-right	`… [ … ]`
	`new` (with argument list)	n/a	`new … ( … )`
	Function Call	left-to-right	`… ( … )`
	Optional chaining	left-to-right	`?.`
19	`new` (without argument list)	right-to-left	`new …`
18	Postfix Increment	n/a	`… ++`
	Postfix Decrement		`… --`
17	Logical NOT	right-to-left	`! …`
	Bitwise NOT		`~ …`
	Unary Plus		`+ …`
	Unary Negation		`- …`
	Prefix Increment		`++ …`
	Prefix Decrement		`-- …`
	`typeof`		`typeof …`
	`void`		`void …`
	`delete`		`delete …`
	`await`		`await …`
16	Exponentiation	right-to-left	`… ** …`
15	Multiplication	left-to-right	`… * …`
	Division		`… / …`
	Remainder		`… % …`
14	Addition	left-to-right	`… + …`
	Subtraction		`… - …`
13	Bitwise Left Shift	left-to-right	`… << …`
	Bitwise Right Shift		`… >> …`
	Bitwise Unsigned Right Shift		`… >>> …`
12	Less Than	left-to-right	`… < …`
	Less Than Or Equal		`… <= …`
	Greater Than		`… > …`
	Greater Than Or Equal		`… >= …`
	`in`		`… in …`
	`instanceof`		`… instanceof …`
11	Equality	left-to-right	`… == …`
	Inequality		`… != …`
	Strict Equality		`… === …`
	Strict Inequality		`… !== …`
10	Bitwise AND	left-to-right	`… & …`
9	Bitwise XOR	left-to-right	`… ^ …`
8	Bitwise OR	left-to-right	`… | …`
7	Logical AND	left-to-right	`… && …`
6	Logical OR	left-to-right	`… || …`
5	Nullish coalescing operator	left-to-right	`… ?? …`
4	Conditional	right-to-left	`… ? … : …`
3	Assignment	right-to-left	`… = …`
			`… += …`
			`… -= …`
			`… **= …`

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

Let's examine some of the issues that may seem different to you accordingly.

Comparison of 3 elements

How can this situation be like this? The 2 look the same and logically the result of the 2 should be true. But this is not the case, why?

```
console.log('3>2>1',3>2>1); //false
console.log('1<2<3',1<2<3); //true
```

First of all, JS does not execute the code as you see it. It looks at the table above. The comparisons are executed as Left to Right →. and it executes it in the following order, advancing the changes Numeric → Bool, Bool → Numeric.

```
//Adım 1
console.log('3>2>1',true>1);
console.log('1<2<3',true<3);
```

```
//Adım 2
console.log('3>2>1',1>1);
console.log('1<2<3',1<3);
```

```
//Adım 3
console.log('3>2>1',false);
console.log('1<2<3',true);
```

Assignment of 3 elements

How come it says 1 here?

```
let a=3; a=5c=1;
console.log(a=b=c); // 1
```

Assignment = Right to Left ← so let's see how the steps work.

```
//Adım 1
b=c // b =1 return 1
console.log(a=b); //
```

```
//Adım 2
a=b // a =1 return 1
console.log(1);
```

Note:

We can multiply variants in this manner. The main issue is to understand how to process the operators in the table above, regardless of whether they are typically operated from Right to Left or Left to Right.

Github Samples

- JS Operator Kod Örneği

References

- Javascript: Understanding the Weird Parts
- MDN Operator Precedence

4. Functions

In this section we will be talking about the following topics

- Function Types.
- How to Define Arrow Function.
- Extras Provided by Arrow Function.

4.1 Function Types

The classifications I have made below are according to me, that is, I have classified these function types in different perspectives, since I cannot handle the subject under a single classification.

4.1.1 Named vs Anonymous Functions

If a function has a name, we call it **Regular/Named Functions**. This is the simplest way we know how to define a function. In the example below, we have defined a function called sum. sum(3,5); When we call it from anywhere, it will add the 2 values we want and return the result.

```
function sum(a,b){return a+b;}
```

If the function does not have a name, we just assign its reference (pointer address) to a variable, we call it **Anonymous Functions**. Here the function has no name, only a reference address. sum(3,5); will work. This time we can do our operations on a variable that holds the reference of the function.

```
let sum=function(a,b){return a+b;}
```

4.1.2 Pure vs Constructor vs IIFE vs High Order Functions

If the function does not hold a state, processes the values it receives from outside and returns the result, we call such functions **Pure Functions**. An important point here is that it should not change the parameters passed to it as values. In other words, it should serve in a very isolated way. In the example below, we call this type of functions Pure Functions because it does not make any changes to the variables a and b during the addition process and only produces a result using them and returns this result.

```
function sum(a,b){return a+b;}
```

If we create an object with a function. We call this type of functions **Constructor Functions**. For example, in the example below, it can be created from the desired users by taking the name and age values.

```
let User=function(name,age){
   this.name=name;
   this.age=age;
}

const ally=new User('Ally',12);
```

If a function is called immediately and the result/result object is stored in a variable, such functions are called **IIFE (Immediately invoked function expression)**. The purpose of this type of functions is to ensure that the variables created in this scope are only accessible in that scope. In general, IIFE method is used in libraries such as jQuery, Prototype to avoid conflicting variable names and to perform operations within the scope. I will elaborate on this topic in more detail in the future with the module concepts I will explain later.

```
const talkOnur= (function talk(name,text){
    var talk='talk';
    return talk+'' + name+': '+text;
})("Onur", "Merhaba");
```

Since Generators functions are one of the new function types that come with ES, I will talk about this topic later.

Functions that take 1 or more than 1 function reference as a parameter in their arguments or return the return value as a function are called **High Order Functions**. Array, which we know very well, is always this type of function that we use frequently.

```
[].forEach (e=> …) //iterate every elements
[].find(e=>…) //return an element according to condition
[].findIndex(e=>…) //return an element index according condition
[].filter(e=>…) return items according to condition
[].sort(e=>…) sorts according to given condition
[].map(e=>…) transform existing array to other array
[].reduce(e=>…) combines elements and return an element
[].some(e=>…) check some elements
[].every(e=>…)
```

If we want to give an example that returns a function as a return value, we can give the following popular example. I want to create a 10 adder. You can use High Order

Functions to create this type of function template.

```
function makeAdder(x){
    return function add(y){
         return x+y;
    }
}
const tenAdder=makeAdder(10);
console.log(tenAdder(12));  //22
```

4.1.3 Sync, Async vs Generator Functions

The JS world used to be very simple. We could get things done with a function running synchronously one after another. The following operation is of the Synchronous Functions type. All lines are executed sequentially.

```
function addAndPrint(a,b){
    const sum=a+b;
    console.log(`Toplam:${sum}`);
}

addAndPrint(3,4)
```

But this type of function will not be enough during the communication between the browser and the server. Many AJAX (communication) requests go from the browser to the server and if we wait for their results synchronously, the user will have to wait continuously in the browser. Instead, we do async operations and create a callback function during the process. This callback function waits for the result of the AJAX request. Then there is a block of code that works according to the response from there. Of course, the code gets very complex, but it allows the end user to deal with other tasks in a very nice way. Nowadays, this is not only a relationship between the browser and the server, there are APIs and services everywhere. For example, when calling an AWS S3, DynamoDB service, every transaction is usually async. Therefore, functions with this type of operation are called **Asynchronous Functions**

```
function wait1SecAndSayHello(callback){
    setTimeout(()=>{
       console.log("Hello")
       return callback({name:'onur'});
```

```
    },1000);
}

function main(){
     const result=wait1SecAndSayHello((result)=>{
        console.log(result);
    }
)
main();
```

In the example above you can see how this is done with callbacks. Callback Hell occurs when these async functions are called one after the other, which makes it very difficult to read the code. Promise, await, async are used to improve this situation.

Finally, I would like to talk about **Generator Functions**. Normally, when we are processing a function, we cannot stop and restart the function in the middle of this process. Thanks to yield, this is now possible. Also Generator functions can return a group of values in the same operation, not a single value.

```
function* generator(i) {
  yield i;
  yield i + 10;
}

var gen = generator(10);
console.log(gen.next().value);
// expected output: 10
console.log(gen.next().value);
// expected output: 20
```

4.2 Arrow Functions ?

We can say that we write our function codes, which we call arrow functions, in a more mathematical language.

```
sum=(a,b)=>a+b;
```

When we explain below how normal functions are defined with arrow functions, you will think that you are writing a math function. You will realize that this way of writing makes it easier for you to write code and increases code readability.

If we look at different Arrow function definitions, I can give the following as examples.

- Constant function that takes no parameters
- Pure function taking a parameter and returning the response
- Definition without parameter encapsulation() because it has only one parameter
- etc...

```
pi=()=>Math.PI; console.log(pi());
sum=(a,b)=> return a+b; console.log(sum(2,7));
square=x=>x*x; console.log(square(9))
f=(x)=>{const y=x+3; return (g=(t)=>t*(t+5))(y)}; console.log(f(2));
```

So how does it make the code more readable? If we give from the **Higher Order Function** above. A function written in the normal way is written as follows

```
function makeAdder(x){
    return function add(y){
         return x+y;
    }
}
const tenAdder=makeAdder(10);
console.log(tenAdder(12));  //22
```

You can see how simple the same code written in Arrow Function is. This is the power of the ES6 Arrow function.

```
makeAdder = (x) => (y) => x + y
const tenAdder = makeAdder(10);
console.log(tenAdder(12));
```

4.2.1 Arrow Function Extras

The following concepts are a bit complicated. I will explain these concepts later in Deep JS topics. I just wanted to mention them briefly here.

Note 1: Arrow functions have no constructor or prototype. It is not used with **new**. Its purpose is not to create an object instance.

Note 2 as arrow functions own **this binding** does not bind itself, the lexical scope context automatically does the binding itself.

5. Type Conversions and Default Parameters

In this section I will focus on the following topics;

- What are Truthy and Falsy values?
- What are Type Coercion, Conversion and Double exclamation points (!!)?
- How does Default Parameters work and what are the advantages?

5.1. Falsy and Truthy

First of all, I would like to start by explaining the concept of "Falsy". We call variables whose value is False in Boolean scope "falsy". These ensure that **if(){},** *** **while(){},** etc. do not enter the relevant code blocks in case comparisons and create logical flows.

There are 7 types of falsy values in Javascipt. All values below are recognized as false by JS.

- **false** : as a boolean value
- **0 :** int as value
- **0n:** as a BigInt value
- `""`**,** `''`**,** ``` **:** Empty string values. Double quotes, Single quotes and String template literals
- **null:** If an assignment is specified such that a variable has no value. let a=null
- **undefined:** If no assignment is made when defining a variable, let a;
- **NaN:** Not a Number

Truthy values are all values except falsy. For example, the following values are always true in a Boolean context

```
if (true)
if ({})
```

```
if ({age:12})
if ([])
if ([])
if (42)
if ("0")
if ("false")
if (new Date())
if (-42)
if (12n)
if (3.14)
if (-3.14)
if (Infinity)
if (-Infinity)
```

5.2. Type Coercion, Conversion and Double exclamation points

Each value also has a Truthy or Falsy counterpart in the Boolean context in Java. When we declare this variable as if (variable) or !!variable object implicitly (in the background), it undergoes a transformation as I mentioned above. This process is called coercion. Double exclamation points are also a separate usage method that runs this coercion process. To explain with a few examples below.

```
let age=39
let name="Onur"
let user={age:39 name:'Onur'}
let users=[]
let school;//Coercion
if(age) true
if(name) true
if(user) true
if(users) true
if(school) falselet rslt=!!school  //Double exclamation point is carved 2 times -1.
console.log(result) //false
```

Conversion can be done implicitly or explicitly and its main purpose is to convert between types. It is called Number → String or String → Number conversion operations. The following examples give examples of implicit and explicit conversions.

```
5+2.0 //7 where the float value is implicitly converted to int.
Number("7") // here the string value is explicitly converted to int.
```

5.3. Default Parameters

There are if checks where we check whether the values passed in JavaScript are always undefined and assign default values to them. In such cases Default Parameters makes your code more beautiful and readable.

```
//Without Default Parameter
function ekle(arr,val){
   if(arr===undefined) arr=[];
   arr.push(val);
   return arr;
}

//With Default Parameter
function ekle(arr=[],val){
   arr.push(val);
   return arr;
}
```

Sometimes there may be situations where a default value is assigned to a variable at the first stage and then this value will be formed from the variables passed in. For example, in Redux store, store default values are passed to the function as default parameters for such cases. You can see that when we pass undefined values in the addition operation below, we use them from the default parameters.

```
function sum(a=10, b=2){
   return a+b;
}

sum (2,2) //4
sum ()    //12
sum (3)   //5
```

Finally, in the example below, you can see that the default parameter only works when no value is passed and undefined is passed. This is not the case for other falsy values.

```
function test(num = 1) {
  console.log(typeof num);
}

test();          // 'number' 1
test(undefined); // 'number' 1
```

```
test('');        // 'string' ''
test(null);      // 'object' null
```

References

- https://developer.mozilla.org/en-US/docs/Glossary/Truthy
- https://developer.mozilla.org/en-US/docs/Glossary/Falsy
- https://developer.mozilla.org/en-US/docs/Glossary/Type_conversion
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters

6. Rest

What advantages does Javascript Rest give us? How does it make the code easier to read?

In summary, I will try to explain the following concepts in this section.

- What is Javascript Rest?
- What are the Advantages and Disadvantages of Javascript Rest?
- How does Rest work in the background?

6.1 What is Rest

The concept of rest is already a concept that is frequently used in other languages we know. For example, I would like to give a *main* example from C, Java which is the beginning of the application.

In the following C program ***argc*** is the number of arguments, argv is the arguments passed in the array. Here, when the program is first run, the parameters to be passed from outside are passed to the main function in this argument array.

```
//Sample C code
int main( int argc, char *argv[] )  {
```

Let's look at the same in the Java programming language.

```
//Sample Java code
public class SampleJava {
    public static void main(String[] args) {
        for(int i = 0; i < args.length; i++) {
            System.out.println(args[i]);
        }
    }
}

//Sample Code
public class SampleJava {
    public static void main(String ...args) {
        for(int i = 0; i < args.length; i++) {
            System.out.println(args[i]);
        }
    }
}
```

It passes these arguments as parameters as soon as the application starts. Here, later on, in Java language, varargs(...) allowed to pass 0 or more parameters as array.

The situation I described above has always been the need to pass an array as a parameter to functions. In the meantime, what advantage does it give us to do this with 3 dots (...), that is, rest, instead of writing it with [] array.

Let's explain this with an example. Let's have a number array and try to find the results of it.

```
function sum(numbers){
    let result=0; numbers.forEach(el=>result+=el);return result;
}
console.log(sum([2,4,6,5])); //17
```

Let's rewrite the code with Rest. As seen in the code below, our function has not changed, but it allows us to pass the parameters to the function as we want without the need for an array object in the calling part.

```
function sum(...numbers){
   let result=0; numbers.forEach(el=>result+=el);return result;
```

```
}
console.log(sum(2,4,6,5))//17
```

6.2 What are the Advantages and Disadvantages of Rest?

If you don't have an array at that stage of the code, it saves you from creating an extra array for it. This gives you an advantage. However, if you want to add parameters before and after this rest. If you want to put other variables in front of the rest in the form of (x, y, ...numbers), this is possible, but if you want to put it after (...numbers, x, y), you cannot do this, which is a disadvantage. When you write the code, you need to write the code knowing these advantages and disadvantages.

6.3 How Rest Works in the Background

So can we do these operations without using rest? Here, even if the function takes no parameters, we can access the parameters we pass as values as arguments. **Array.prototype.slice.call** creates an array object with the arguments passed as values in this section.

```
function sum() {
    const numbers = Array.prototype.slice.call(arguments);
    let result=0; numbers.forEach(el=>result+=el);return result;
}

console.log(sum(2,4,6,5))//17
```

What are the arguments passed as values? Array is a map object that can be transformed

```
{ 0:2,1:4,2:6,3:5 }
```

Actually, the algorithm written in the background is as follows ... if there is a point, extract the values before it in the argument object and convert it to an array.

7. Spread

What advantages does Javascript Spread give us? How does it make the code easier to read?

In summary, I will try to explain the following concepts in this article.

- What is the Javascript Spread Concept?
- Spread Usage → Function:Function.prototype.apply()
- Spread Usage → Array: push(), splice(), concat(),
- Using Spread → Object and Array Copy
- Using Spread → Math: Math.max()

7.1 What is Spread

Javascript Spread is the array spread assigned to this section. In a nutshell

```
...[2,4,6,8] => 2,4,6,8
```

So this time you have array values in the background and you want to distribute them like values outside the array.

7.2 Function Call/Apply Samples

Let's examine the use of Javascript call and apply. The call function allows you to call the parameters one by one with the given context, while apply allows you to do this as an array.

```
function sum(x,y,w,z){
    let result=x+y+w+z; return result;
}

console.log(sum.call(this,3,5,7,9)); //24
console.log(sum.apply(this,[3,5,7,9])); /24
```

With Spread, we have been able to do the above operation directly with the call. array spreading feature has made this possible.

```
console.log(sum.call(this,...[3,5,7,9]));
```

7.3 Array Use Cases

Array.push()

Array allows us to add the elements we want at the end. Here we can give a single element as well as multiple elements. You can use spread instead of apply function here.

```
const arr=[2,3]; arr.push(4,5) console.log(arr);//[2,3,4,5]
const arr=[2,3]; arr.push.apply([4,5]) console.log(arr);
const arr=[2,3]; arr.push([...4,5]) console.log(arr);
```

Array.splice() Samples

If we want to add another array to the middle, end and beginning of the array, spread is very advantageous here. it has become more readable than doing it with splice.

```
const arr=[1,2]; arr.splice(1,0,4,5); console.log(arr)//[1,4,5,2]
const arr=[1,...[4,5],2]; console.log(arr)
```

Array.concat() Samples

If you want to concatenate an array, you can do it with concat, but with spread it is quite easy and legible.

```
const arr=[1,2]; console.log(arr.concat([3,4],[5])) //[1,2,3,4,5]
console.log([...[1,2],...[3,4],...[5]]);
```

7.4 Array and Object Cloning Samples

In the examples below you can clone the array and objects with the spread operator.

```
const arr1=[1,2]; const arr2=[...arr1]; arr1.push(3);
console.log(arr2);
const obj1={a:1,b:2}; const obj2={...obj1}; obj1['c']=3;
console.log(obj2);
```

Object Copy Methods (Shallow/Deep Copy)

During **Shallow copy**, only the values in the first-level hierarchy are copied. Spread operator also works according to this copy.

Deep copy, on the other hand, traverses all sub-hierarchies and copies each value, which is laborious and tedious, but deep copy is mandatory so that references do not affect each other.

Shallow Copy

```
let obj = { a: [1, 2, {users: [{ name: ‘onur’, age: 12 }] }], b:2 };
const cloneObj={…obj}; //same => Object.assign({}, obj);
obj.a[2].users[0].name=’deniz’;
obj.b=4;

console.log(cloneObj);//Result interesting..
//cloneObj.b is 2 not 4 because it clones first level and cloneObj holds a new memory loca
tion for b
//but cloneObj.a[2].users[0].name is 'deniz' why because it hold reference cloning is not
deeply.
```

Deep Copy

1. Serialize/Deserialize : For fast copying - you can use JSON.parse/stringify.
2. lodash.clonedeep

```
const clonedeep = require('lodash.clonedeep')
```

3. Recursively you can copy the object by assigning them to each other.

```
function clone(obj) {
     if (obj === null || typeof (obj) !== 'object' || 'isActiveClone' in obj)
        return obj;    if (obj instanceof Date)
        var temp = new obj.constructor(); //or new Date(obj);
```

```
    else
        var temp = obj.constructor();    for (var key in obj) {
        if (Object.prototype.hasOwnProperty.call(obj, key)) {
            obj['isActiveClone'] = null;
            temp[key] = clone(obj[key]);
            delete obj['isActiveClone'];
        }
    }
    return temp;
}
```

7.5 Math Samples

Using math.max is similar to using apply above. Instead of apply you can use spread operator for large array.

```
console.log(Math.max(1,5,3)); //5
console.log(Math.max.apply(this,[1,5,3])); //5
console.log(Math.max(…[1,5,3])); //5
```

8. Destruction Assignments

Destruction When we need access to the variables in the structures that we create (packaged) in an object or array, in order to access these variables more easily, instead of the assignment / assignment method we know, a concept called Destruction has been created with a concept called Destruction, fast and easier extraction methods from the package. This allows the code to be both more readable and shorter.

For simplicity and clarity, you can see in the example below how easy it is to access the name and surname in a props object. You can see the same in the array operation. It allows us to make assignments that we would do in many lines one by one in a single line.

Object Deconstruction

```
props={ name:’Hello’, surname:’World’}
state={ age:12} . //Deconstruction
const {name,surname}=props; //>> const name=props.name
```

```
const {age}=state; //>>const age=state.age
console.log(`My name is ${name} ${surname} and ${age} years old`)
```

Array Deconstruction

```
colors=[‘red’,’green’,’blue’];
let [val1,val2,val3]=colors //>> let val1=colors[0] ...
console.log(val1,val3) //red blue
```

8.1 If We Examine Object Destruction Operations in Detail

8.1.1 Assigning Different Names to Destruction Variables

You need to use componentWillReceiveProps when using React when you have this.props and nextProps objects, that is, when you have more than 1 of the same object at the moment.

```
const props={ name:'Hello', surname:'World'}
const {name:adi,surname:soyadi}=props;
console.log(`Adım ${adi} ${soyadi}`)
```

8.1.2 Assign Default Value

You may not receive a variable during destruct, but you can assign default values to variables so that your system is sensitive to it. For example, in the example below, surname may not exist and by assigning a default value to it from the beginning, we prevent it from being undefined. Of course, if the value existed, it would use it.

```
const props={ name:'Hello'}
const {name,surname="Unknown"}=props;
console.log(`My Name ${name} ${surname}`)
```

8.1.3 Destruction operation when passing Function as Value

This method is also very useful. You have an object and only certain values of it are meaningful for this function. Doing this fragmentation while passing this object makes

the code easier to use.

```
function sayHello({name,surname,other:{age}}){
    console.log(`My name is ${name} ${surname} and age ${age}`)
}
const props={
    name:'Deniz',surname:'Dayibasi',other:{age:8,height:128} }
sayHello(props);
```

If you have a user array with a similar feature, you can do this during the loop.

```
for (const {name: n, surname: s, other{age:a}} of people) {
```

8.1.4 Using Rest

You may only want to deal with certain variables and throw the rest into rest.

```
const props={name:'Deniz',surname:'Dayibasi',other:{age:8, height:128} }
const {name,...rest}=props;
console.log(rest)
```

8.2 If we examine Array Destruction Operations in detail

8.2.1. Make assignments by skipping certain indexes

In the example below, values can be assigned without val2. Intermediate values can be omitted with a ,, sign.

```
colors=[‘red’,’green’,’blue’];
let [val1,,val3]=colors //>> let val1=colors[0] ...
console.log(val1,val3) //red blue
```

8.2.2. Default Value Assignment

```
const colors=['red','green','blue'];
let [val1,,val3,val4='orange']=colors
```

```
console.log(val1,val3,val4) //red blue orange
```

8.2.3 Rest

```
const colors=['red','green','blue'];
let [val1,...rest]=colors
console.log(rest) //green , blue
```

Comments

As you can see from the examples above, you can see that it abstracts more logical parts from the developer while minimizing the code writing. Many topics within the scope of Functional Programming, which I will explain in the future, are all about this. By abstracting from the developer, it is aimed that developers will write more stereotypical code in the future.

I use this type of code development needs a lot during React. I think it shortens the code and makes it more readable.

References

- MDN Destructuring assignment

9. Template Literals

Since it is easier to explain some things through examples, I will start by giving an example. Let's have a function called sayHello. In this example, let's print this information to the console. By defining String templates in the following way, we can create a result string with the variables we want and use it wherever we want.

```
sayHello(name, age){
   console.log(`My name is ${name} and I am ${age}  years old.`)
}
sayHello('Onur',39) //Output->My name is Onur and I am 39 years old
```

Strings are one of the most used variable types in our programming. Especially in the Web Development part of UI development, browsers work with DOM (Document Object

Model). And DOM models can create String, we used to use tools like the one below, which we use a lot.

- String replace
- Mustache.js
- Handlebars.js

In the DOM part, these started to be resolved in JSX with React, and those in String started to be resolved in Template Literals. Since our topic is Template Literals, I will talk about the structures that use String as a template on String (Text).

- String Definition Methods
- Uses and Purposes of Template Literals

9.1 Types of String Definitions

Until now, there were 2 ways to define a String: single quotes or double quotes

```
const name='Onur' const surname="Dayibasi"
console.log(name+" "+surname)
```

But why can we define 2 types? Because there are situations where we need to nest 2 different strings during web development. I know the following is not a very meaningful example, I just added it to show how it is done technically.

```
const desc='Onur "Dayibasi" and I am 39 years old'
console.log(desc)
```

If a string is defined with single quotes, it cannot contain single quotes or double quotes if there are double quotes, the escape character \ is used for this.

```
const desc='Onur \'Dayibasi\' and I am 39 years old'console.log(desc)
```

When we look at the string concatenating methods (Concatenating), we can concatenate all text, numbers, etc. with +.

```
const my={name:'Onur',surname:'Dayibasi',age:39}
const desc='My name is '+my.name+' '+my.surname+" and age is "+my.age;
console.log(desc)
```

If we want to define the code with a new line or tab, we need to give \t \n definitions directly in the string.

```
const my={name:'Onur',surname:'Dayibasi',age:39}
const desc='My name is '+my.name+' '+my.surname+" \n age is "+my.age;
console.log(desc)
```

9.2. Uses and Purposes of Template Literals

New String definition method => `backsticks` . this way we can do what we did above

- **Interpolation:** Embedding variables into text

```
const a=5;
const b=7;
console.log(`sum of a and b is ${a+b}`);
//sum of a and b is 12

const my={name:'Onur', surname:'Dayibasi'}
console.log(`My name is ${my.name} ${my.surname}`);
//My name is Onur Dayibasi
```

- **Multiline** \t \n Instead of these characters for a new line or tab, you can define codes like this

```
const string = `Ali say Hello!
                Veli say How Are you
                asas                 is awesome`;
console.log(string);
```

- **Template Tags:** If you want to create much more complex tag processors. You can use these Template Tags to define structures like DSL (Domain Specific Language) that will parse and operate the generated text according to different defined logic.

For example, the sampleTags function below creates a template like this and you can use these templates elsewhere in the code.

```
function sampleTag(strings,name, surname) {
   return  `${strings[0]}${name[0]}...${strings[1]}${surname[0]}...`
}
const name='ONUR';
const surname='DAYIBASI';
const result=sampleTag`${name} and surname is${surname}.`;
console.log(result);//Result 'My name is O... and surname is D...'
```

Comments

In this type of string definitions, abstractions have come to the code that enable the user to write the code in a single way as in other ES6 features by allowing you to do the string operations that we do explicitly in the code with an abstraction. In the future, when I talk about Functional Programming, I will explain what is intended with these structures more and more.

References

- MDN Strings
- MDN Template Literals

10. Enhanced Object Literals

How to define an object in Javascript? Enhanced Object Literals explains how this process is simplified.

In this topic, I would like to summarize the following topics.

- How do we define an object?
- How Enhanced Object Literals provides advantages.

10.1 How Do We Define Object?

Defining a Javascript object is quite simple.

```
{} //empty Object
{name: "Onur" , age:39} //key value pairs
```

Here, when we want to give these values as variables from outside, we need to define the object as follows.

```
const name="Onur"; const age=39;
console.log({name:name, age:age});
```

In the example below you can see how the object is created in the function that gets the variables from outside. You need to define key-value pairs inside the object all the time.

```
const name = "Onur"; const age = 39;//Default Object Initiation
callFunc = (fn, name, age) => {fn({ fn:fn, name: name, age: age })}
callFunc(console.log, name,age);
```

Let's develop this situation a little more. In case we say that the dynamic variable name should be taken from outside, you need to define the object as a prop.

```
const name = "Onur"; const age = 39; h='height';//Default Object Initiation
callFunc = (fn, name, age, hProp, height) => {
  const obj={ fn:fn, name: name, age: age};
  obj[hProp]=height;
  fn(obj);
}
callFunc(console.log, name, age, h, 180);
```

10.2 What are the advantages of Enhanced Object Literals?

As you can see in the code below, we can define the object without a key-value pair. This makes the code more readable.

```
const name = "Onur"; const age = 39; h='height';
//Enhanced Object Literals
```

```
callFuncEnhanced = (fn, name, age) => {fn({ fn, name, age })}
callFuncEnhanced(console.log, name, age);
```

In our other example, when we use [h] during object initialization, we can define the object in a very simple way.

```
const name = "Onur"; const age = 39; h='height';
//Enhanced Object Literals
callFuncEnhanced = (fn, name, age, h, height) => {fn({ fn, name, age,[h]:height});}
callFuncEnhanced(console.log, name, age, h, 180);
```

Referances

- MDN Object

11. Loops and For..of method

To summarize this topic;

- How many different types of looping methods there are and basic concepts.
- For, While, Do/While Loops
- forEach and Other Ready Functions.
- Why is For...of Needed? Are Other Loops Not Enough?
- Recursive Functions

11.1. How Many Different Types of Looping Methods Are There and Basic Concepts

You can use the following loop methods in JS. But I will explain why there are so many different loop methods below.

- for
- while
- do/while
- forEach

- for...in
- for...of
- recursion

Continue/Break commands are a way of managing the loop extra. For example, you have logical operations but you don't want the code to run for the relevant part of the loop. If you want it to skip **continue.** If you say that you have finished your operations in the part where the loop is located and you do not need it to work for others, you can make your loop work more efficiently by using the **break** command.

Recursive: Creating loops with self-calling functions and terminating the loop in a certain situation is realized. If the function does not stop calling itself at some point, your code will give a stackoverflow error.

iterator: In languages such as Java, C++, Collection etc. are structures that enable navigation in data structures. In Collection data structures derived from Iterable interface, iterator is a pointer that holds which node it is on. Similar structures are also available in JS.

11.2. For, While, Do/While Loops

In procedural programming, Basic, Pascal, C and Java we use looping methods all the time. These loops are used for very general purposes. Thanks to the flexibility provided, you can create any type of loop you want in a very simple way.

Features of the For loop;

- You set the condition from the beginning, so this loop may never start (i<5)
- You set the variable that determines the condition in the loop. (i)
- In the loop, you specify the initial value of the variable and how it will increase in for. (i++), that is, it is said to increase by 1 each time.
- The for loop shape is determined from the beginning.
- Continue/break commands work.

```
for(let i=0;i<5;i++) console.log(i);
```

Characteristics of the While loop;

- You specify the condition again from the beginning (i<5)
- The initial value of the variable is not set in the loop

  . (i=1)
- The increment of the variable can increase, decrease or change as desired within the scope of the loop.
- It provides a more flexible loop structure than for.
- Continue/break commands work.

```
let i=1; while(i<5) {console.log(i); i++};
```

Features of the Do/While cycle

- The condition checks the probe this time (i<5)
- It always does the first operation.
- The other parts are the same as **while loop**.
- Continue/break commands work.

```
let i=1; do{ console.log(i); i++} while(i<5) ;
```

11.3. forEach

Array, Map and Set data structures have a function like forEach that can go over all elements. The disadvantage is that there is no possibility to continue/break.

```
const numberArray=[1,2,2,4,5];
const colorSet=new Set(["red","green","blue"]);
const colorMap=new Map([['red',1],['green',2]])numberArray.forEach(el=>console.log(el))
colorSet.forEach(el=>console.log(el))
colorMap.forEach((value,key, map)=>{console.log(`${key}:${value}`)})
```

I would like to talk about other functions similar to forEach using Array, which are implemented with Functional Programming (functional programming) High Order Functions method. → map, filter, find, findIndex, reduce, every, some, etc. functions are ready-made functions provided to developers combined with loops for a specific purpose.

- **forEach:** return all array elements
- **filter:** return all array elements and filter by condition
- **map:** map all array elements to another array.
- **find:** find the element of interest in the array.
- **reduce** array to produce a single output.
- **every** check if all given elements pass the test.
- **some** is whether some of the given elements passed the test

I will elaborate on these issues in more detail when I explain functional programming in the future.

11.4. For…of

In fact, you can provide any kind of loop you want with the loop types mentioned above. Why forEach, for...of, for...in etc. loops appeared?

There are different data structures in applications. These are String, Array, LinkList, Set, Map, Trie Tree, Graph etc. When you want to navigate these data structures, we need easier methods than the loops mentioned above. These types of structures improve both the code writing and the readability of the code.

In the following data structures

- Arrays
- Strings
- Maps
- Sets
- DOM data structures
- Symbols

I have made a few examples of this below. A very important point here is that you can use continue/break commands in for...of.

```
const arr=[1,2,2,4,5];
const str='12245'
const set=new Set([1,1,2,4])
const map=new Map([['a',1],['b',2],['c',3]]);for(let val of arr) console.log(val);
for(let val of str) console.log(val);
for(let val of set) console.log(val);
for(let val of map) console.log(val);
```

Here I would also like to briefly mention the for...in loop. For example, you have an object and you want to return its property, you can use for...in for this.

```
const me={name:"Onur", age:39, height:180};
for(let key in me) console.log(key+":"+me[key]);
```

There are other methods of accessing the object's property key or entry. Object.keys() and Object.entries() methods can create loops on the results returned as an array

```
const me={name:"Onur", age:39, height:180};for(let key of Object.keys(me)) console.log(ke
y)
for(let keyVal of Object.entries(me)) console.log(keyVal)
```

11.5. Recursive Functions

These self-calling functions are a very common technique, especially in functional programming. Calling the function itself makes the code more readable and shorter, but as I mentioned above, you need to be careful about stack-overflow.

In the example below, the fibonacci number is the sum of the previous fibonacci number and itself. F(n)=F(n-1) + F(n-2)

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
```

If we write this in a recursive way, we get an image closer to the mathematical definition.

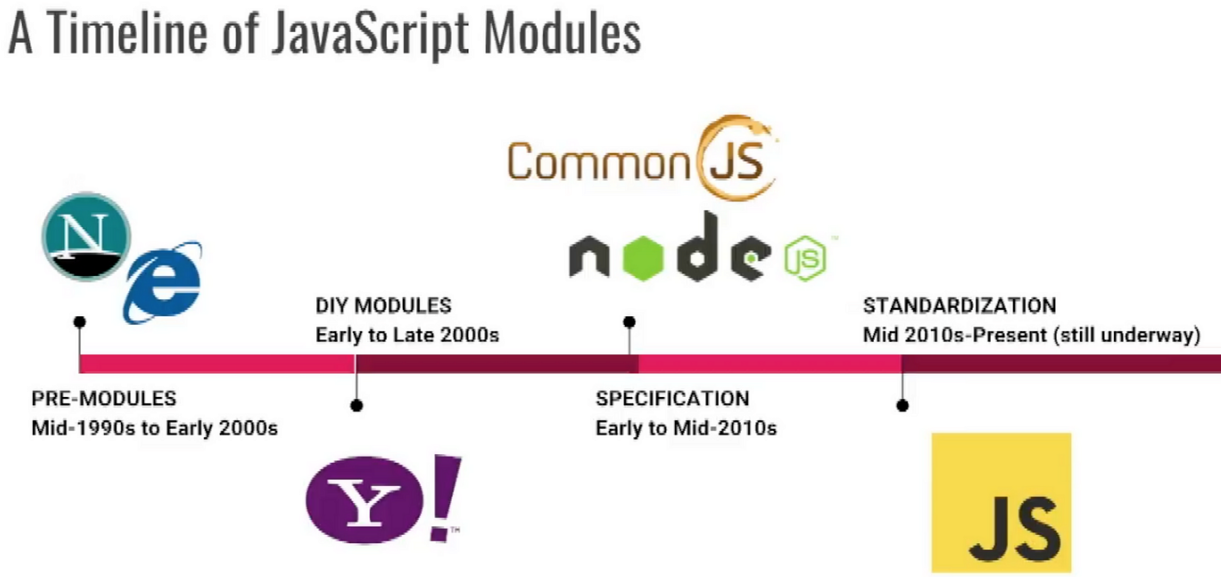
```
const fibonacci=(num) => {
   if (num <= 1) return 1;
   return fibonacci(num - 1) + fibonacci(num - 2);
}
console.log(fibonacci(5));
```

12. Module (IIFE → CJS → AMD → ES6)

I will try to explain concepts such as How the Module Structure of Javascript Developed, What is Module, IIFE, (Common.JS,NPM), (AMD, Require), Import, Export.

- How did Javascript Module Structure Evolved?
- Pre-Modules Era (inline-scripting)
- DIY Modules Era (IIFE)
- NodeJS Era (Common.JS - Sync)
- Browser Module Era (AMD, Require.JS - Async)
- ES6 Module Era

12.1. What was the evolution of the Javascript Module Structure?



from JavaScript Modules Past & Present

- **Pre-Modules Period:** The period when Javascript is used with inline-script and script tags. The period when small animations and color changes are made and web pages are prepared. The period when all kinds of variables and functions are defined in global scope.
- **DIY-Modules Period:** The period when the first Javascript libraries Yahoo UI, JQuery, utility libraries emerged. In this period, you can see that the first module abstractions were made. IIFE (Immediately Invoked Function Expression) method was used for this.
- **NodeJS Era (Common.JS - Sync):** JS can run on Local and Server machines. With NodeJS, JS has increased its momentum and needs. It was necessary to develop the concept of direct module in NodeJS Common.JS . Since there is a need to access the files on the local machine during this process, Sync loading is sufficient. If the module is in the cache, it is accessed by reading directly from the local file. (How does NodeJS support import?)
- **Browser Module Era (AMD and Require.JS - Async):** These developments on the server side naturally triggered the browser side. However, since the browser has

a network, modules had to be loaded as **async** and a standard had to be created for this. Async Module Definition and Require.JS solved these problems.

- **ES6 Modules:** JS has standardized the solution methods that took place in its ecosystem over time. With **import**, **export** keywords, it has made these operations the most simple, readable and understandable.

12.2. Pre-Modules Era?

In the **pre-modules** (mid-90s, early 2000), the competition between IE and Netscape was going on. What you could do on browsers was very limited. In this process, the use of JS was very low and you could do all your inline-scripting operations with script tags in a single html (index) file.

```
<html>
<body>
   JS Script<script>
    var sum=3+6;
 </script><div id="msg"></div>
   <div id="sum"></div><script>
    document.getElementById("sum").innerHTML = sum;
</script>

</body>
<script>
   document.getElementById("msg").innerHTML = "Merhaba Dunya";
</script>
</html>
```

Sonuç

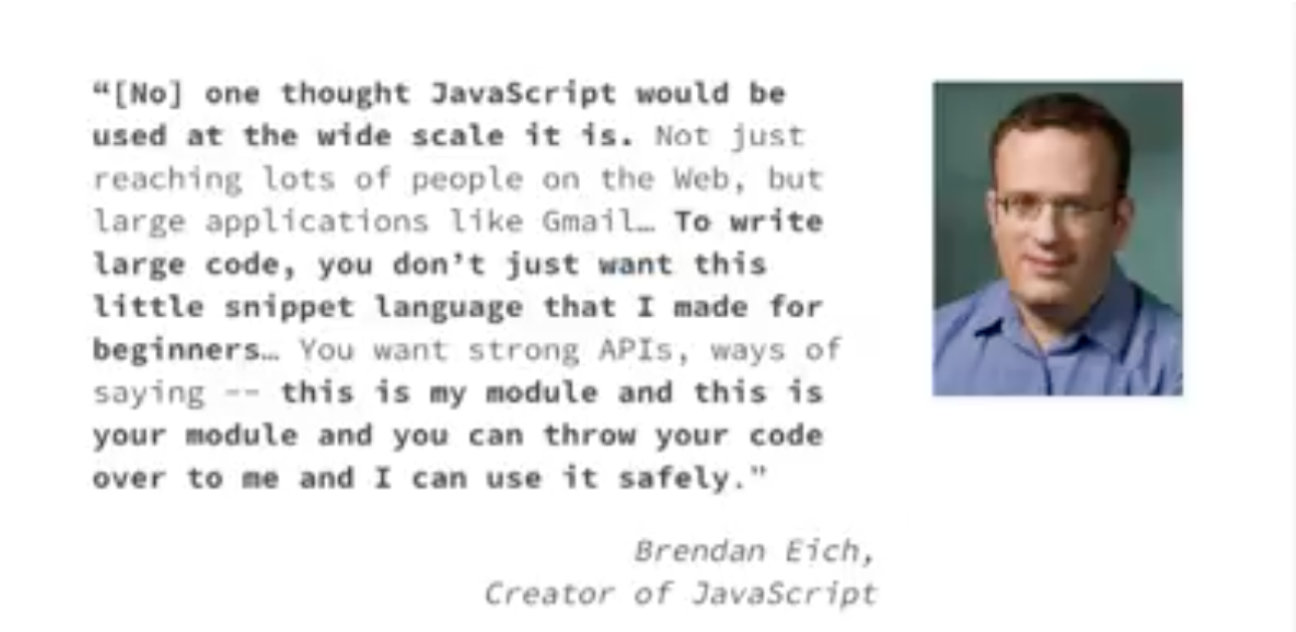
Another use of the script tag is to separate index.html from other js information. For example, instead of the small code fragments above, I can collect the tim js codes in a single file. For example, I can collect all the codes in the **index.js** file. Or I can divide them into different files like **DOMOps.js, MathOps.js**.

In my opinion, Modularity has already been realized by dividing HTML, JS and CSS files into separate files with the first part. In the following processes, JS codes were divided into separate files according to their logic, allowing the code to be readable and written on a larger scale.

```
<script src=”URL”>
```

Is this structure suitable for developing large-scale software?

Brendan Eich, who created JS directly, gave the answer to this question. No one thought that Javascript would be used on a large scale in this way. They targeted beginners with such small code pieces. He mentioned that strong APIs and module system are needed to develop big code like Gmail etc.



Brendan Eich ideas about JS

So how did such a simple language become what it is today, perhaps the most widely used language in the world, and meet the needs of different types of developers? Maybe it is wrong to call it a simple language because Javascript was developed inspired by Scheme and Self languages, which include Functional Programming, which I will explain later.

You can see that the features of the Self programming language are based on prototype-base inheritance, which I have marked in bold.

> About Self

> Self is a **prototype-based dynamic object-oriented** programming language, environment, and virtual machine centered around the principles of **simplicity, uniformity, concreteness, and liveness**. Self includes a programming language, a collection of objects defined in the Self language, and a programming environment built in Self for writing Self programs. The language and environment attempt to present objects to the programmer and user in as direct and physical a way as possible. The system uses the **prototype-based style of object construction.**

> Self contains a **user interface and programming environment** designed for "serious" programming, enabling the programmer to create and modify objects entirely within the environment, and then save the object into files for distribution purposes. The metaphor used to present an object to the user is that of an outliner, allowing the user to view varying levels of detail. Also included in the environment is a graphical debugger, and tools for navigation through the system.

> You can see that the parts of the Scheme language that I have marked in bold form the basis of the Functional Programming

> features High Order Functions, etc. that come with ES6.

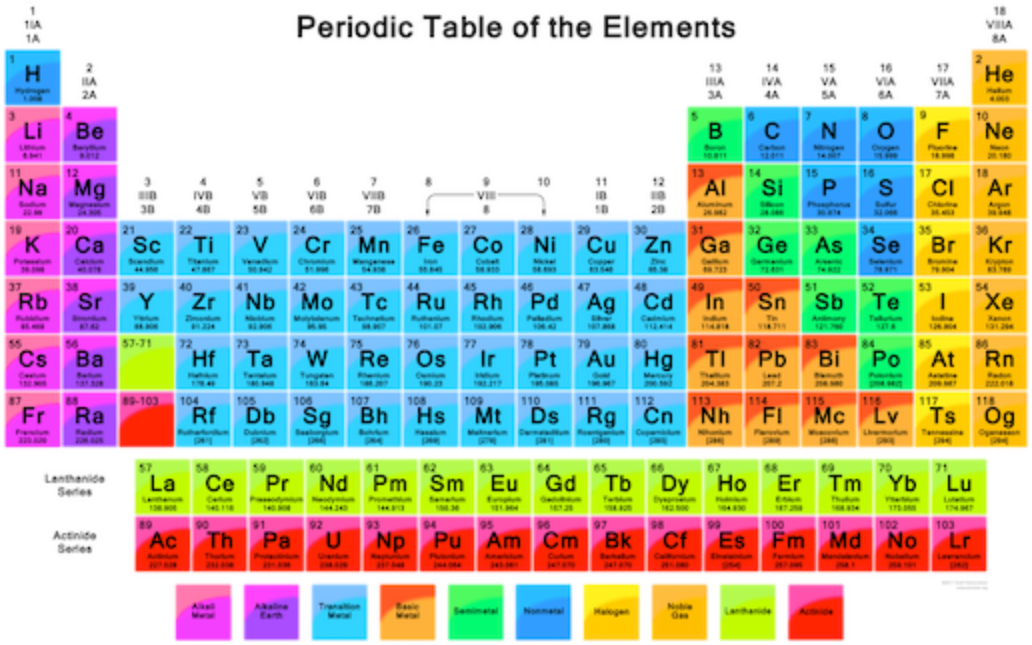
> Scheme is a general-purpose computer programming language. It is a high-level language, supporting operations on structured data such as strings, lists, and vectors, as well as operations on more traditional data such as numbers and characters. While Scheme is often identified with symbolic applications, its rich set of data types and flexible control structures make it a truly versatile language. Scheme has been employed to write text editors, optimizing compilers, operating systems, graphics packages, expert systems, numerical applications, financial analysis packages, virtual reality systems, and practically every other type of application imaginable. Scheme is a fairly simple language to learn, since it is based on a handful of syntactic forms and semantic concepts and since the interactive nature of most implementations encourages experimentation. Scheme is a challenging language to understand fully, however; developing the ability to use its full potential requires careful study and practice.

What was the problem? What were the problems and challenges during this period?

- Code reusability
- Managing dependency on other code
- Maintenance challenges

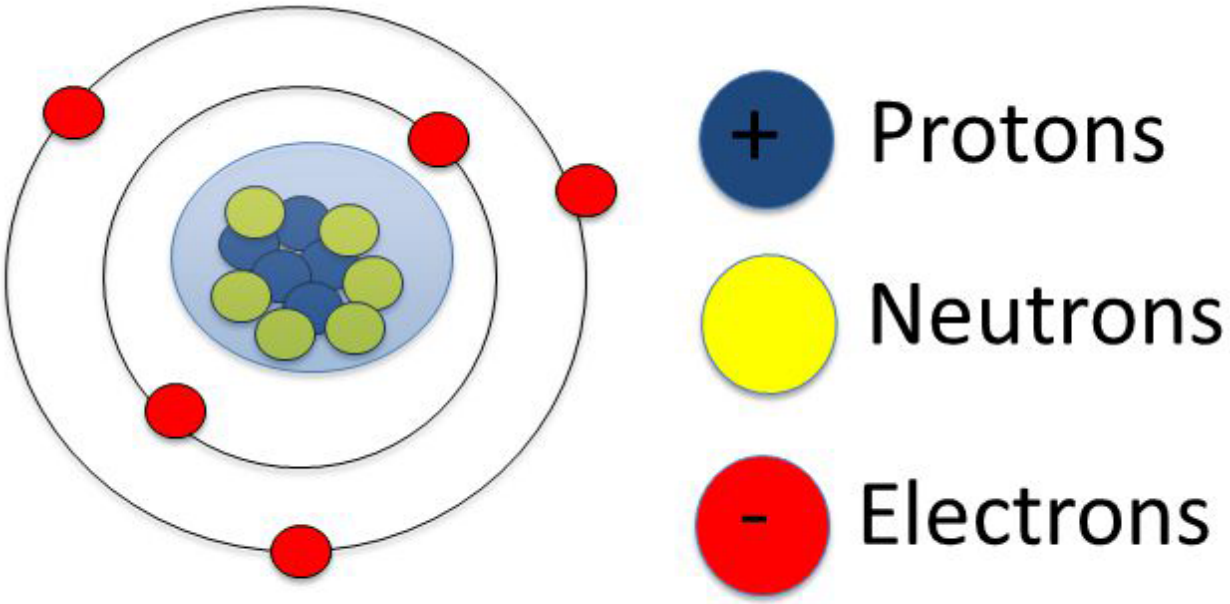
12.3. DIY Modules Era (IIFE)

What is a Module? I liken modules to atoms and molecules. When we examine these concepts we call atoms, there are different different atoms. We can think of these atoms as modules that do not need other components on their own.



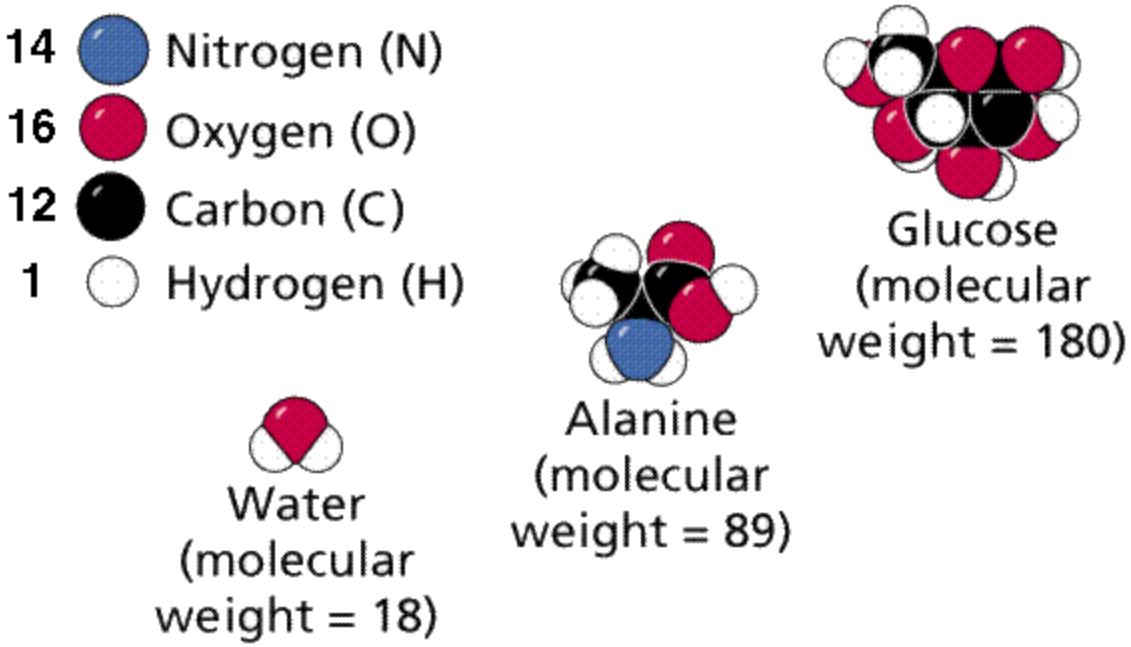
Periodic Table

When we look inside these atoms, they consist of a nucleus made up of protons and neutrons and electrons circulating around this nucleus.



Atom Structure

and from their combination molecules are formed. These molecules combine with each other to form things, systems or organisms.



Molecule Structure

Code modules are similar to this. It is possible to create blocks of code and link them together to build large systems such as libraries, applications, etc. So what is a module?

- It performs its own internal logical operations.
- Provides data and functions to the outside through the interface.
- They are structures that can use the code modules they need from outside.

DIY Modules . In the 2000s Yahoo was quite powerful in the Internet world. It was more powerful than Google. Google had just been founded in 1998 and Yahoo engineers said that **Global variables cause bad results.** They created a manifesto for them.

- Because the ubiquitous accessibility of global variables meant that someone could change the value you gave at any time in the code.
- It means that others use functions that no one else should see or use.
- Sometimes it meant conflicting code or variable names of 3rd party libraries.

They used **IIFE(Immediately invoked function expression)** to realize these abstractions.

What is **I**IFE? I mentioned all function types in the Functions section, and I mentioned that I will explain IIFE in more detail in the module section in the future.

> If a function is called immediately and the result/result object is stored in a variable, such functions are called IIFE (Immediately invoked function expression). The purpose of this type of functions is to ensure that the variables created in this scope are only accessible in that scope. In general, IIFE method is used in libraries such as jQuery, Prototype to avoid conflicting variable names and to perform operations within the scope. I will elaborate on this topic in more detail in the future with the module concepts I will explain later..

Here, with the first libraries, the concept of **Namespacing** was created instead of thousands of Global variables. Let there be one big Global Object belonging to Yahoo,

belonging to JQuery and access everywhere through this object and the variables inside this object cannot be accessed directly from outside. Yahoo or JQuery objects have become libraries that hold and manage the state of the entire page on their own.

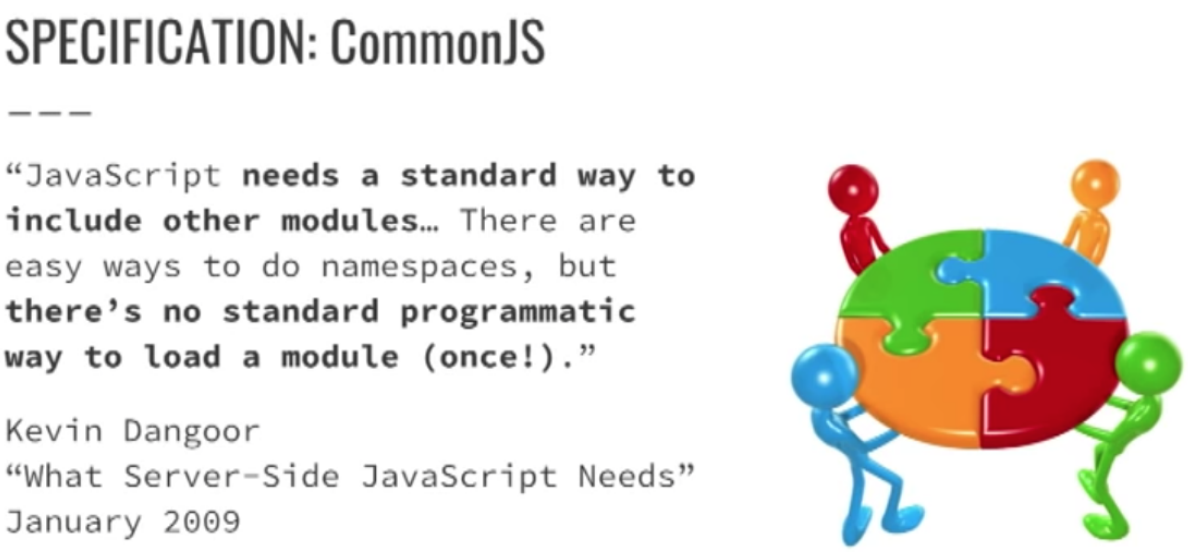
In the code block below, I access the counter from outside as I want.

```
var myModule={ counter:0}
myModule.counter++;
console.log(myModule.counter); //1
```

In the code block we wrote with IIFE, we closed the counter to external access. We started to manage it from the state return function without creating a global state. This is what early UI libraries did. Managing global variable and function usage.

```
var myModule=(()=>{ var counter=0;
    increment=()=>counter++;
    print=()=>console.log(counter);
return {increment:increment,print:print}
})();

myModule.increment();
myModule.print(); //1
```

12.4. NodeJS Era (Common.JS - Sync)



Spesification

In 2009, with Node.JS, they thought that there was a possibility to use Javascript inside the server and that there was a **great potentiall** here. As a result, it was possible to develop a fullstack WebApp with a single language only with JS.

These Enterprise needs, that is, the need for a more standard programmatic module loading with code in a different way from the browser of JS modules in order to develop larger projects with JS.

DipNote: Here, browsers only run JS, PHP, JSP, ASP, Ruby, etc. languages were eventually producing JS, HTML for Web-side applications, but the Backend became increasingly standardized. The Controller - Service - Data layer and the logics here have turned from code development to configuration management by moving towards REST → GraphQL. UI turned into complex structures with more logical operations and components. This situation started to create both the fullstack concept and JS Web Stack structures. Reducing both developer costs and maintenance costs would be more in the interest of the bosses.

At this stage, JS gained incredible momentum in Web Application development. The same applies to Python for artificial intelligence and its derivatives. After all, a group

doing a Python AI project can build the entire technology stack on Python and develop the Web application in Python.

In his blog post "What Server Side JavaScript needs", Kevin Dangoor has collected the requirements to create an ecosystem in the future in a recommendation post.

- Cross-interpreter standard library, creating standard libraries that JS executors (Rhino, Spidermonkey, V8 and JSCore) use in common.
- Standard interfaces
- **Standards for modules to include each other**
- Code packaging, versioning standards
- Package repository (like NPM)

If we look at what the standards are for modules to include each other in this section; CommonJS aims to establish the loading mechanism and server-based design logic as follows.

- **import :** require() method
- **export:** module.exports object

To explain with a simple example. We connect modules with ***require(aws-sdk*)** and create functions that provide services to the outside ***module.exports = { purgeQueue}***. The whole structure is based on this.

```
const AWS = require('aws-sdk');
AWS.config.update({ region: 'eu-west-1' });
const sqs = new AWS.SQS({ apiVersion: '2012-11-05' });



//=============================================================================
const purgeQueue = (queueURL, fnCallback) => {
    var params = {
        QueueUrl: queueURL
    };
    sqs.purgeQueue(params, function(err, data) {
        if (err) fnCallback(err)
        else fnCallback(data);
    });
}

module.exports = {
    purgeQueue
}
```

AWS-SQS

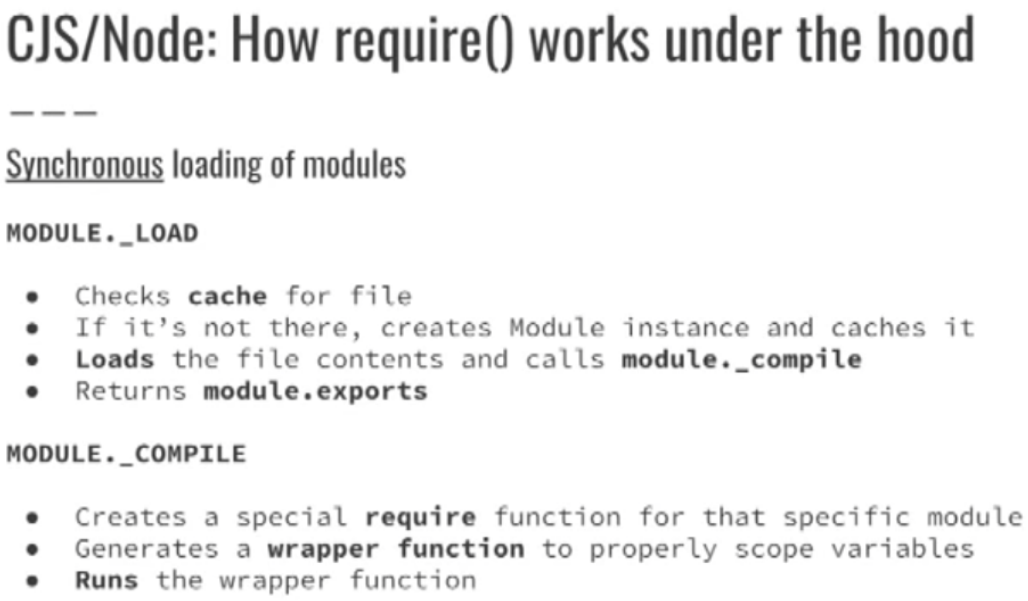
Here is how require works. Require is actually a function and performs sync load operation. As I mentioned at the beginning, async needs are not a priority at this stage since Node needs to access JS in the local filesystem.

- **Module._LOAD :**

  First it checks the cache. If not, it starts module creation process, loads the file and

  MODULE._COMPILE

  function. It creates a scope wrapper function like IIFE and returns module.exports. Actually the mechanisms are the same, they just specify the specification and abstract this part away from people and developers don't have to think about it.



CommonJS Require

Wrapper function below;

```
(function(exports, require, module, __filename, __dirname) {
// Module code actually lives in here
});
```

Require Wrapper Fonksiyon

12.5. Browser Module Era (AMD Require.JS - Async)

Although there were a number of mechanisms running CJS on the browser side, it did not solve the main problem on the browser side. As a result, JS files that needed to be uploaded to the user's computer over the network had to be downloaded and uploaded Async. It had to work more optimized.

AMD (Asynchronous Module Definition) was created for this. And Require.JS Module Loader was created. The purpose of this is as follows

> RequireJS is a JavaScript file and module loader. It is optimized for in-browser use, but it can be used in other JavaScript environments, like Rhino and Node. Using a modular script loader like RequireJS will improve the speed and quality of your code.
>
> *IE 6+ ..... compatible* ✔️
>
> Firefox 2+ ..... compatible ✔️
>
> Safari 3.2+ .... compatible ✔️
>
> Chrome 3+ ..... compatible ✔️
>
> Opera 10+ ..... compatible ✔️
>
> *Get started then check out the API.*

```
//Calling define with module ID, dependency array, and factory function
define('myModule', ['dep1', 'dep2'],function (dep1, dep2) {

//Define the module value by returning a value.
returnfunction () {};
});
```

12.6. ES6 Module Era

Finally, can we gather so many different Module loading methods and specifications under a single standard. They thought about how we can offer **IIFE, CJS, AMD** methods directly in JS Standards

- The **built-in** module structure in JS should be like this.
- Supported by **all browsers**
- **Asynchronous (Async)** should work

When we examine ES Module Syntax, we can easily provide variables and functions (In JavaScript, functions are first-class objects) that we want to share with other modules thanks to **export**. Thanks to **import**, we can use the variables or constants we want from other modules. You can use it as follows.

export

```
// msg.js
export default const msg = 'Yay ES6!';

// lib.js
export const sum = (a, b) => a + b;
export const product = (a, b) => a * b;
export const quotient = (a, b) => a / b;
```

import

```
import msg from './msg.js';

import * as lib from './lib.js';
import { sum, product } from './lib.js';
```

ES6 Module Syntax

- export
- export default
- import * as from
- import {def1, def2} from
- import def1 from

So how do we do this if we want to use it in HTML and not in JS. For this, you can use **module** as type in the script tag.

```
<script type="module" src="main.js"></script>
```

References

- JavaScript Modules Past & Present **Mejin Leechor**
- https://requirejs.org/

13. NPM

This article is actually the second of the Modules article and since I wrote this article before, I wanted to edit it a little more. NPM building blocks are actually created by grouping and packaging modules for a specific purpose. We can call NPM building blocks JS libraries. The reason why the JS ecosystem is so big is due to the power of NPM and OpenSource code development culture.

In this article, I will summarize the following topics.

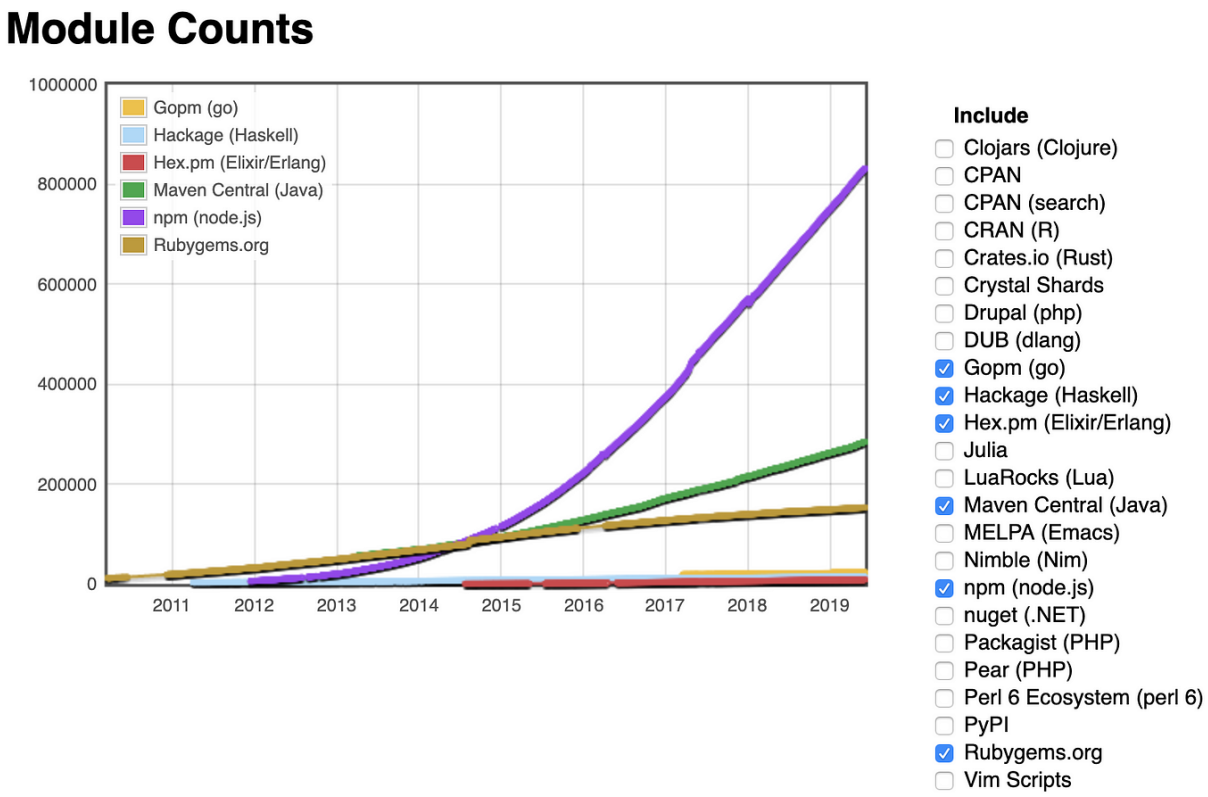
- What is NPM?
- How Does NPM Work?
- How to Use NPM?
- What are the Differences Between Bower, NPM, Yarn?

13.1. What is NPM?

NPM (Node Package Manager), as the name suggests, is a node module manager created to save and use node modules in a common place. Over time, node modules have become available in frontend libraries thanks to tools such as webpack and babel. Javascript can be used in browser, server, developer and desktop applications.

Previously, Javascript could only run web pages on the browser with a single library (YUI, Mootools, ExtJS, JQuery) and their plugins. With the progressive developments, this has become insufficient for larger projects and now there is a need for structures where many modules, large and small, use each other. For this, these modules need to be found in a common place and used from there. npm has provided this need.

In fact, this type of module sharing needs existed in other languages and has been used since long before. For example, you can see the development of package management tools such as Maven, Gopm, Rubygems.org etc. at http://www.modulecounts.com/. Of course, the number and increase of these modules does not show the quality of these packages, but it is an important source in terms of showing the interest in languages.



13.2. How does NPM Work?

Hosting over 1 million JS packages, npm is dependent on hundreds of thousands of modules when developing next generation projects, how can we manage this dependency and their updates. This is where SemVer (Semantic Versioning) comes into play.

- https://docs.npmjs.com/about-semantic-versioning
- https://blog.npmjs.org/post/162134793605/why-use-semver

Semantic versioning makes it possible to manage the dependencies of so many libraries on each other. For example, the number 3 in version 1.5.4 means. When using a library, how changes in that library affect your component.

- **major (breaking):** When you make an update due to a change in the module interface, you will need to make some updates to your code.
- **minor (feature):** This is when a new feature is added without a change in the module interface.

- and **patch** versions **(fix)**:**: We advance this number when you only solve a bug without a change in the module interface.

In the example below you can see how New Product, Patch release, Minor Release and Major Release are incremented.

Incrementing semantic versions in published packages

To help developers who rely on your code, we recommend starting your package version at `1.0.0` and incrementing as follows:

Code status	Stage	Rule	Example version
First release	New product	Start with 1.0.0	1.0.0
Backward compatible bug fixes	Patch release	Increment the third digit	1.0.1
Backward compatible new features	Minor release	Increment the middle digit and reset last digit to zero	1.1.0
Changes that break backward compatibility	Major release	Increment the first digit and reset middle and last digits to zero	2.0.0

NPM Semantic Versions

12.3. How to use NPM?

After creating a user account on the pm page and confirming it, via console/terminal

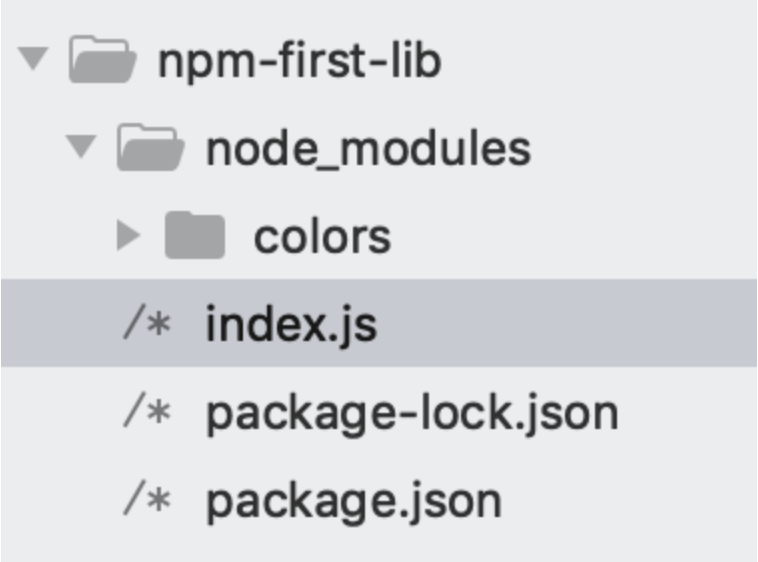
```
npm login
Username: <your username>
Password: <your password>
```

After performing npm authantication, you can check for which user you are logged into the npm system with the command below.

```
npm whoami
```

The next step is to write a simple module. For this, the following npm example is quite simple

Console is developing a module with a simple interface that prints logs. The only function is printMsg() . it always prints a fixed message. First of all, npm init and npm install colors - save will create the following files and modules.



node package structure

- **package.json:** Contains basic information such as package name, version, dependencies etc.
- **package-lock.json:** Holds the versions and integrity information of other modules that this package depends on.
- **node-modules:** Downloaded modules are downloaded under this package.

13.3.1. Publish as step 1.0.0.

Here we are trying to write a function that prints the message on the red colored screen, but since we are writing read, this function cannot perform its operation completely.

```
const color=require(“colors”);exports.printMsg=function(){
 console.log(“this message from npm-first-lib”, “read”);
}
```

We publish the first version of the module with the following code to the terminal.

```
npm publish
```

Later, when we download and use this module npm install npm-first-lib from another project, we see that the code just writes a message to the screen.

13.3.2 Step (correcting a mistake in the module) Patching as 1.0.1

We fix the code and make sure that the message printed on the screen is printed in red colors.

```
var color=require(“colors”);

exports.printMsg=function(){
 console.log(“this message from npm-first-lib”.red);
}
```

When we run the commands below after fixing the code, you can see that the patch number has increased by 1. The important thing here is to know that the patch only fixes bugs and does not change the interface used at all.

```
npm version patch
npm publish
```

13.3.3 Step (Adding new feature in module) 1.1.0 new feature

It has an interface that does the same thing, but we have added a new capability where we can perform the console print operation with the color we want when we give color from outside.

```
var color=require(“colors”);
```

```
exports.printMsg=function(printColor){
 if(!printColor) printColor=”red”;
 console.log(“this message from npm-first-lib”[printColor]);
}
```

In this case we make a minor change. We advance the 2nd digit and publish the module again.

```
npm version minor
npm publish
```

13.3.4 Step (Change module interface) 2.0.0 new breaking

We want our function to take msg as a variable, not just color as a variable. In this case, applications using this module will not work when they receive updates.

```
var color=require(“colors”);

exports.printMsg=function(msg, printColor){
 if(!printColor) printColor=”red”;
 console.log(msg[printColor]);
}
```

Therefore, you can use versions like npm install 1.1.x or 1.x without upgrading your updates to the next level. If the modules you are using have major updates like below, you need to change your application according to the new interface of the component.

```
npm version major
npm publish
```

13.4. What are the Differences Between Bower, Npm, Yarn?

So what is Yarn with React or Bower that we used before? These are JS package managers, but their periods and purposes are different.

For example ***Bower*** Frontend JQuery, Masory etc. we were manually downloading other frontend libraries on index.html and putting them under a package and linking

them from index.html. After Npm installs Bower on the system, instead of manually downloading these packages, we use bower and load them under bower_components via bower.json.

Afterwards, the relevant link can be given as follows. or Grunt, RequireJS and related libraries can be installed on the system.

```
<script src="bower_components/jquery/dist/jquery.min.js"></script>
```

After npm and Yarn became available in the modern frontend, the trend has shifted from bower to these 2 package managers. npm was released in 2011, Yarn is a tool that runs on npm developed by Facebook to fix some problems with npm.

- Speed
- Security
- Offline

Yarn has been developed to manage project dependencies in a better way with *yarn.lock* file besides speed and security events and therefore it has been preferred in many places. After NPM version 5, Yarn incorporated similar features into its structure and solved the dependency problem in a similar way in *package-lock.json.

Therefore npm and yarn are still the most preferred package managers.

14. Map/Set Data Structures

In this chapter

- I will talk about the data types that JavaScript provides.
- Object was enough for us, why did we need Map type?
- Array was enough for us, why did we need Set type?

14.1. What Types of Data Does Javascript Provide?

JS has different types to hold different types of values and perform different operations on these values.

- To perform mathematical operations, to perform date time operations, to hold textual information such as a person's name, address, etc., true or false

the types of values we want to keep in each of them are different. JS developers

- **Primitive types:** defines types like string, number, boolean, null, undefined, object function, symbol.

> **Note 1:** different types have different methods. string.toLowerCase() number.toFixed() object.hasOwnProperty and these primitive types are automatically converted to their Object types (String, Number, Boolean, Object, BigInt, Null, Undefined, Function, Symbol)

> **Note 2:** Javascript is a dynamic typing language. that is, you can assign different types to the same variable. you cannot do this in java, c etc. languages with static type checking. JS dynamic typing has a lot of advantages. I will talk about this in the following topics. I will talk about the advantages in the Functional Programming section in the future. With Typescript, you can use the JS language with static typing.

```
var val = 1000;    // number assign
val     = 'abc';   // string assign
val     =  true;   // boolean assing
```

Therefore, you can do type checks during coding to understand what type the content of the value is. You can use the **typeof** keyword for this.

```
typeof sss => string
typeof true => boolean
typeof 10.00 => number
typeof undefined => undefined
```

```
typeof {age: 15} // "object"
typeof function sum() {} // "function"
typeof undefined // "undefined"
typeof Symbol('x') // "symbol"
typeof NaN  // "number" (Not a Number) is a number presentation
typeof [1,2,3] => object
typeof null => object
```

You cannot use this method for Map, Array, String, Boolean, Date, etc. You can use the **instanceof**. method for these, but this method does not support primitive types.

```
new String("aaa") instanceof String  // true
"aaa" instanceof String              // false
"aaa" instanceof string              // Reference Error..
```

For type check, you can make a type check via the constructor to snippet a control you will make through the **constructor** to cover all of them.

```
const is = (type, val) => ![, null].includes(val) && val.constructor === type;is(Array,
[1]); // true
is(ArrayBuffer, new ArrayBuffer()); // true
is(Map, new Map()); // true
is(RegExp, /./g); // true
is(Set, new Set()); // true
is(WeakMap, new WeakMap()); // true
is(WeakSet, new WeakSet()); // true
is(String, ''); // true
is(String, new String('')); // true
is(Number, 1); // true
is(Number, new Number(1)); // true
is(Boolean, true); // true
is(Boolean, new Boolean(true)); // true
```

14.2. Object was enough for us, why did we need Map type?

I mentioned the concept of Object in the previous sections. Defining Object is quite simple and we can create the map data structure we want in the form of Key/Value. So why did we need **Map** data type instead of **Object**. What can't Object do?

- Only String/Symbol can be given as Object key value.

- We use the **for...in** method to iterate through Object elements. Another method is to iterate through Object.keys, Object.values or Object.entries methods
- Problem accessing Object elements in order.
- We were checking with undefined to see if there is a value or not.
- Set and Get were directly accessing the object by treating it as **array** or **.**.
- We had to write a function for the number of object elements.

```
Object.size = function(obj) {
    var size = 0, key;
    for (key in obj) {
        if (obj.hasOwnProperty(key)) size++;
    }
    return size;
};
```

As it can be understood from the above, developers needed a Data Structure API, even though we can use it like an object map. When we examine the Map API;

```
new Map() =>constructorget(key) => function that gets the value corresponding to key. Here
key can be the desired type. Number, Obj etc...

has(key) => does key have a value?

set(key,value) => function that assigns the value corresponding to key. Here key can be th
e desired type. Number, Obj etc...size => number of elements

delete(key) => deletes the data map with key

clear() => deletes all keys and data

keys(), values(), entries() => returns related types as array.for (let pair of mapObj) {
//Loop
    var [key, value] = pair;
    console.log(key + " = " + value);
}
```

14.3. We had enough of arrays; why did we need Set types?

As you know, Array is a data structure that can be defined simply in JS like Object.

```
const arr=["yaz","kis","kis","bahar","ilkbahar"]
```

But it is not a set. Sets do not contain repeating elements. You can provide unique with **.filter** to provide this state array. But this means iterating over the whole system after adding an element.

```
const arr=[“yaz”,”kis”,”kis”,”bahar”,”ilkbahar”]
var unique =arr.filter((value,index,self)=> self.indexOf(value) === index)
console.log(unique);
console.log([…new Set(arr)]);
```

Instead, the **Set** data structure already offers us this possibility. Set API

```
new Set() =>constructorhas(value) => key değeri var mı?size => eleman sayısıdelete(value)
=> elemanı silerfor (let value of setObj) { //Loop
    console.log(value);
}
```

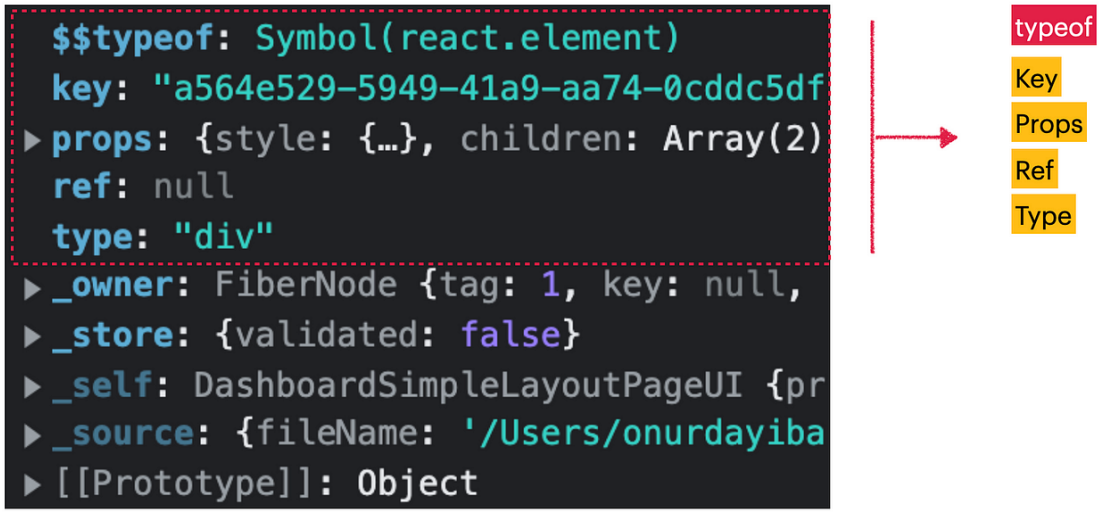
One important conversion here is Arr → Set, Set → Arr conversion should be very simple. Below you can see how simple these 2 are done in a single line.

```
[…new Set(arr)]
```

15.Symbol

We talked about data types in JavaScript, with ES6, the Symbol type was added to the primitive data types. In this article, we will write about why we need the Symbol type.

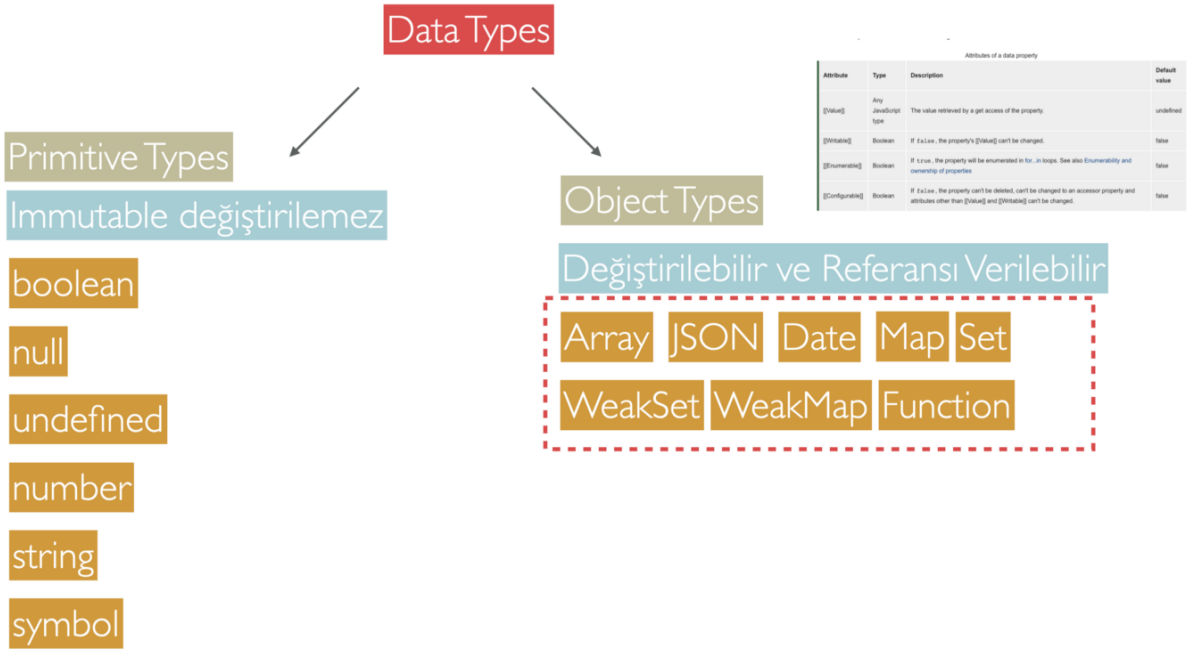
For example, the typeof variable in React objects, which we use the most, is used as Symbol. Why?



React

Tabi öncelikle tip değişkeni varken neden **$$typeof** var diye merak edenler için linkini buraya ekleyeyim → (React Elemanlarında Neden $$typeof Özelliği Var?). Bu yazıyı okuduğunuzda Symbol veri tipinin kabaca güvenlik için kullanıldığını görebilirsiniz.

ES6'da Symbol adında bir tip geldi. Öncelikle JS'de 2 tip tipten bahsediyorduk, birincisi Primitive tipler, Immutable olanlar ve Object tipinde olan Object, Array, Function vb. mutable yani içeriği değiştirilebilen tipler. (JS Data Types) adlı bir makale paylaşmıştım.



JS Data Types

The important issue here is that the content of primitive types is **(immutable)**. In other object types, the most important feature is that they are **key/value string pair whose content can be changed, that is **(mutable)**.

These key definitions in objects are always defined as strings and the flexibility to change objects over string structures is very high.

In Meta Programming, we can perform self-modification, assigning or defining a new property to an object or deleting an existing property via Object[...]. Just like you can see in the example below. I can change the structure of the existing Object as I want.

```
const obj={
  name:'onur',
  sayHello:()=>console.log('hello'),
}

obj['fullname']=obj['name'];
delete obj['name'];
delete obj['sayHello'];


console.log(obj)
```

```
{ fullname: 'onur' }
```

Well, can we assign number or boolean values other than String as the key value of our object?

As you can see, it does not define it. He didn't like the number 2 as a key.

```
const obj={
  name:'onur',
  2:'veli,
}

console.log(obj)
```

```
SyntaxError: Unterminated string constant. (3:4)

  1 | const obj={
  2 |   name:'onur',
> 3 |   2:'veli,
    |     ^
  4 | }
  5 |
  6 | console.log(obj)
```

Well, if we say obj[2] and make a definition, this time it accepts it but converts it to String value. You can find this automatic conversion topic in my article (Type Coercion and Conversion).

```
const obj = { name: 'onur' };
obj[2] = 'veli';
console.log(obj);
```

```


{ '2': 'veli', name: 'onur' }
```

In this case, we need to give String while defining the key values of the object.

Here, even though different types of objects have a **_key_ field ***name***, there is no feature that distinguishes them from each other.

```
const user = { name: 'onur' };
const car = { name: 'bmw i8' };

console.log(user.name);
console.log(car.name);
console.log(user['name']);
console.log(car['name']);
```

```

'onur'
'bmw i8'
'onur'
'bmw i8'
```

Symbol is actually a type that provides us with a unique identifier. So when we want to have a unique value for object keys and not to be confused with other values, string does not provide this.

```
console.log('surname' === 'surname');
console.log(Symbol('surname') === Symbol('surname'));
```

```
true
false
```

Surname ve Symbol..

Here we use the string parameter in Symbol to make this created Symbol object description and more debuggable. Here you could create this symbol empty.

```
console.log(Symbol());
console.log(Symbol('surname'));
```

```
Symbol()
Symbol(surname)
```

Symbol Creation without parameters

15.1 Unique Key Generation

Instead you need to create key values consisting of Symbols.

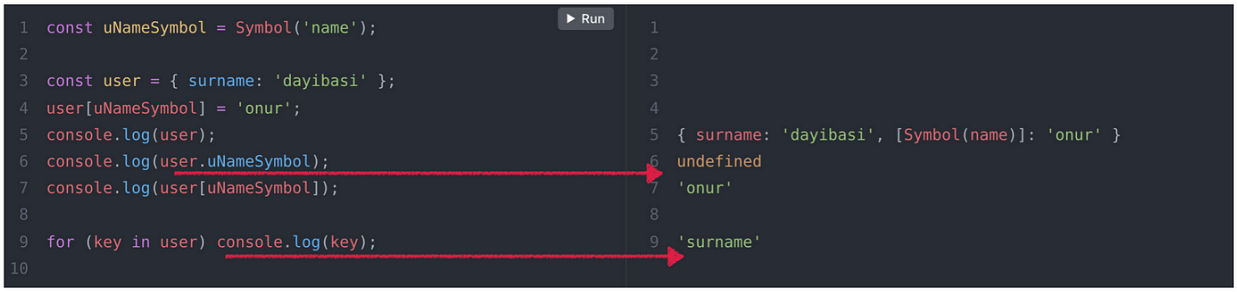
```
const uNameSymbol = Symbol('name');
const cNameSymbol = Symbol('name');

const user = {};
user[uNameSymbol] = 'onur';
console.log(user);

const car = {};
car[cNameSymbol] = 'Bmw i8';
console.log(car);

uNameSymbol === cNameSymbol;
```

```


{ [Symbol(name)]: 'onur' }


{ [Symbol(name)]: 'Bmw i8' }

false
```

Above, I was able to uniquify the key signature in each different type of object, which makes it possible to generate key values that do not conflict with each other when we make object extensions.

15.2 Making Property (Key) Private

As you can see from the screenshot below, key values given as Symbol are no longer accessible as object.__.



2ndly, when we want to get the key elements of the user object, it does not iterate here.

15.2.1 Does it provide complete privacy?

So let's say another library created the object and we are using it, can it hide some information from us?

No, you can get Symbol ***property/key*** definitions with other commands.

15.2.2 So is there an easier way to access Symbol References? (Global Symbol Registry)

Are we going to create symbol references and pass them as parameters somewhere to manage them?

```
function write2Console(obj, nameSymbol) {
  console.log(obj[nameSymbol]);
}

write2Console(user, uNameSymbol);
```

```

'onur'
```

Using Symbol Passing as a Parameter

To use the Global Registry to use these references, Symbol.for() returns a Symbol reference from the Global Registry for the value passed as the key parameter here.

```
const uNameSymbol = Symbol.for('name');
const u2NameSymbol = Symbol.for('name');

console.log(uNameSymbol === u2NameSymbol);
console.log(uNameSymbol === Symbol.for('name'));

console.log(Symbol.keyFor(uNameSymbol));
```

```


true
true

'name'
```

In the same way, we can get the key value in **Symbol.keyFor(symbol)**. In this way, you can access key, Symbol matching through a single Global Symbol Registry.

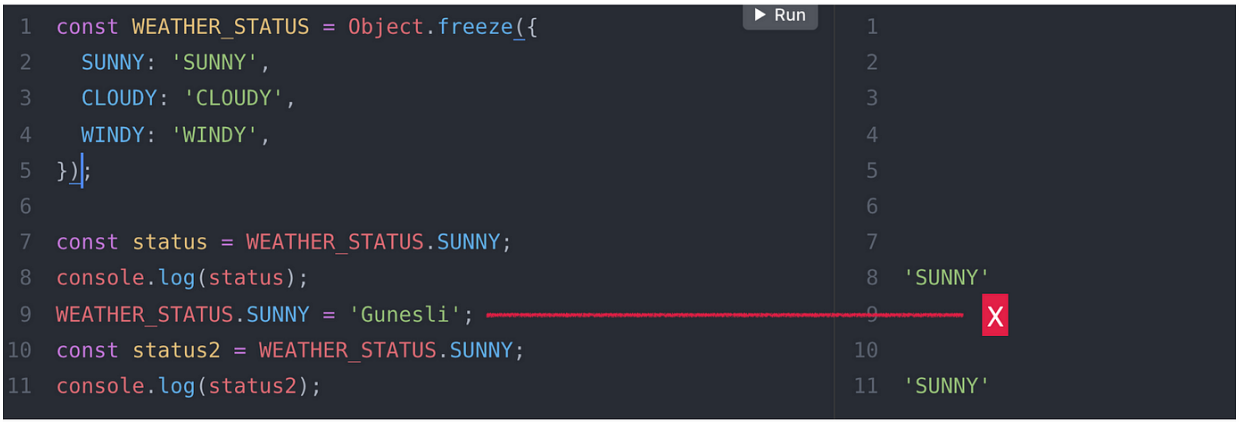
15.3 Make an Enumeration

I created a Const **Enumaration Object,** but the fact that I am creating it does not prevent me from changing the enum values in this object.

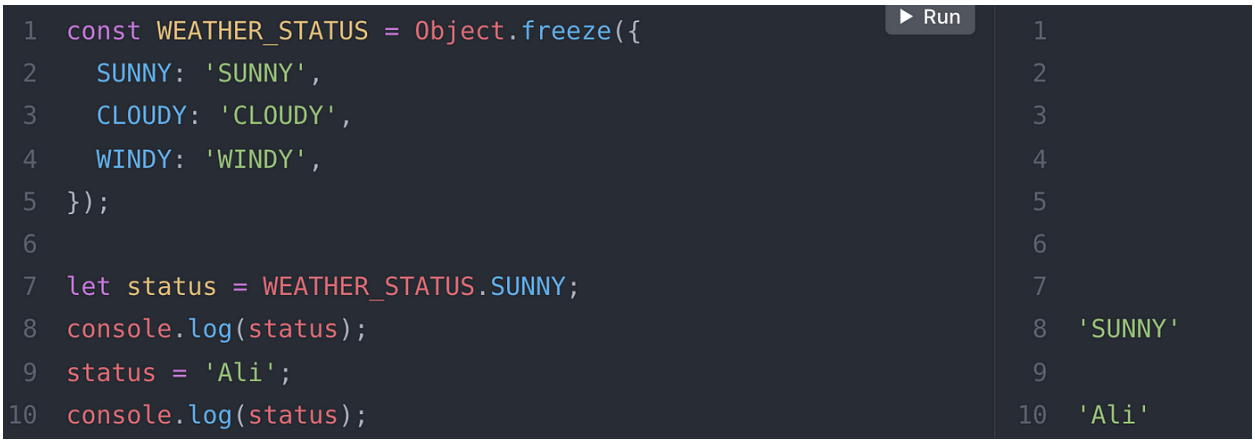
```
const WEATHER_STATUS = {
  SUNNY: 'SUNNY',
  CLOUDY: 'CLOUDY',
  WINDY: 'WINDY',
};

const status = WEATHER_STATUS.SUNNY;
console.log(status);
WEATHER_STATUS.SUNNY = 'Gunesli';
const status2 = WEATHER_STATUS.SUNNY;
console.log(status2);
```

```



'SUNNY'


'Gunesli'
```

So what can be done for this? We can freeze object property modification with Object.freeze.



Does it mean we can't provide this status object a value other than Enum? Sadly, it doesn't. You can assign as follows, however since JS is not a type-based structure, you must utilize TypeScript and Flow transpiler structures.



Does it prevent several enums from assuming the same value? Not at all? It does not stop several enums from having distinct values.

```
const WEATHER_STATUS = Object.freeze({
  SUNNY: 'SUNNY',
  CLOUDY: 'CLOUDY',
  WINDY: 'WINDY',
  ABCD: 'SUNNY',
});

let status = WEATHER_STATUS.SUNNY;
console.log(status);

status === WEATHER_STATUS.ABCD;
WEATHER_STATUS.SUNNY === WEATHER_STATUS.ABCD;
```

```
'SUNNY'

true
```

This is an issue that the **Symbol** type can help us solve. Given that we are making the values unique for enumeration, the definition that follows would be more appropriate.

```
const WEATHER_STATUS = Object.freeze({
  SUNNY: Symbol('SUNNY'),
  CLOUDY: Symbol('CLOUDY'),
  WINDY: Symbol('WINDY'),
  ABCD: Symbol('SUNNY'),
});

let status = WEATHER_STATUS.SUNNY;
console.log(status);

status === WEATHER_STATUS.ABCD;
WEATHER_STATUS.SUNNY === WEATHER_STATUS.ABCD;
```

```
Symbol(SUNNY)

false
```