The Development Of JavaScript Through Time (pdf)

> **Last Updated Date:** 09.05.2023
>

1.1 Beginning

JS Historical Development is separated into two sections.

- **First,** there was the **Browser Era, when JavaScript attempted to exist within browsers.
- **Second:** The ES6,ES7,ES8,ES9,ES10,ES11,ES12 periods following EcmaScript 5, when JavaScript was possible to operate outside of browsers with Node.JS and on servers and desktops.

1.2 The Browsers Era

Brenden Eich: is the creator of the JavaScript programming language. He developed it in 1995 while working for Netscape, which is now Mozilla. Over time, his name was changed to **Mocha LiveScript JavaScript**.

Doug Crockford: JSON (JavaScript Object Notation) was created in 2002. This format, which is a subset of XML, serves as the foundation for both directly supporting JavaScript in the language and interacting with the server.

Jesse James Garrett: defined **Ajax** in a 2005 paper. The web application's underlying communication technique was async communication with the server.

John Resig: In 2006, he created the **JQuery** library. These were libraries that provided developers with an abstraction that eliminated any browser incompatibilities, as popularized by libraries like Prototype, Dojo, and Mootools.

HTML5: Flash is coming to an end. Another breakthrough for JavaScript occurred when technology developers such as Steve Jobs and Mark Zuckerberg stated that they would not support Flash devices in browsers and that the future was in HTML5, CSS3, and JavaScript.

With the start of this trend, all plugins that worked in browsers vanished. Java's Applet and JavaFX, Adobe Flash, and Microsoft Silverlight have all but vanished.

1.3. ES Period (ES5, ES6, ES7, ES8, ES9, ES10, ES11, ES12)

NodeJS was created in 2009 by Ryan Dahl. This infrastructure, which was constructed by forking Chrome's JavaScript compiler, enables JS to run as a server, that is, outside of the browser. And from then on, JS began to run everywhere. It began to appear everywhere, no longer fitting into the browser that brought it into existence, and the committee began to grow indefinitely. The requirements began to diverge. With this distinction, the language began to be remade in an evolutionary manner.

EcmaScript (ES): The underlying standard for JavaScript. Since its inception as a browser-based language, this standard has incorporated Flash's ActionScript and Microsoft's JScript. But eventually the best features of other languages were incorporated into JavaScript, and they vanished.

TC39: The ES Development committee is made up of several members from browser developers and large Web-related companies.

Version: Every year, generally in June, a new edition of ES is published, and each year is designated by doubling the previous year's number. For example, language features announced in 2015 are referred to as ES6, language features published in 2016 are referred to as ES7, and language features deployed in 2019 are referred to as ES10. The JS language changes published in 2015, ES6, were extremely different and advanced the language.

I've included the developments in the JS language based on EcmaScript versions below;

1.3.1 ES6 (2015)

- let and const
- Arrow Functions
- Classes
- Default Parameters
- Template Literals
- Destructing Assignments
- Enhanced Object Literals
- For-of Loop
- Promises
- Spread operator
- Set/Map
- Generators

1.3.2 ES7 (2016)

- Array.prototype.includes
- Exponentiation operator (**)
- Array.prototype.find and Array.prototype.findIndex
- Object.getOwnPropertyDescriptors

1.3.3 ES8 (2017)

- String padding
- Object (values,entries)
- Async Functions
- Shared Memory And Atomics

1.3.4 ES9 (2018)

- Async Iteration
- Rest/Spread
- Promise.prototype.finally

- Regular Expression Improvements

1.3.5 ES10 (2019)

- Array (flat, flatMap)
- Object (fromEntries)
- Optional catch binding
- String (trimStart, trimEnd)
- Symbol (description)
- stable Array (sort)
- Well-formed JSON (stringify)
- Static Field

1.3.6. ES11 (2020)

- BigInt
- Dynamic import
- Nullish Coalescing
- Optional Chaining
- Promise.allSettled
- String#matchAll
- globalThis
- Module Namespace Exports
- Well defined for-in order
- import.meta
- private fields(#)

1.3.7 ES12 (2021)

- Numeric Separators
- String.prototype.replaceAll
- Promise.any() and AggregateError
- Logical Assignment Operators
- Private Class Methods and Accessors

1.3.7 ES13 (2022)

- Await operator at the top-level
- Class field declarations
- Private methods and fields
- Static class fields and private static methods

- RegExp Match Indices
- Ergonomic brand checks for private fields
- .at() function for Indexing
- Temporal function

2. 2015's JS language improvements (ES6)

2.1 let, const

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

What is the purpose of the var?

Variable is referred to as var. We store names, messages, numbers, and dates in these variables and conduct actions on them in the software.

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

So what would happen if we never used it as we mentioned below. First of all, you cannot define age; alone. JS needs to recognize it as an assignment so that it can operate. For the others (name, height) JS will create a variable named name in the global scope and assign it.

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

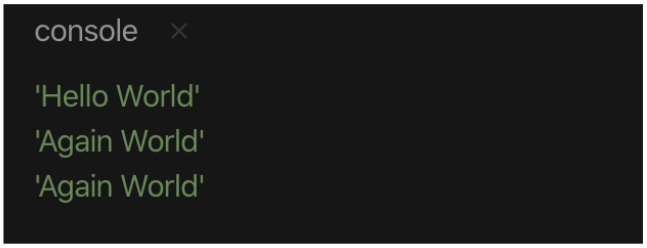
As developers, they don't want other people to be able to access the values they save in functions. They just want that function level of processing. Because the var variable is not utilized on the left in the example below, we can see a foo variable that may be used outside of the function.

var kullanmanın farkı

However, the variables we declare should always exist, be visible, and be accessible inside the defined scope. This is what we developers always aim for. As the amount of code increases, it becomes impossible to manage accesses outside of this and it is unclear where the variable is modified from.

A variable declared in the higher scope must be accessible to and used by those in the lower scope. This is a crucial requirement. Already, this is the desired outcome.

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



Üst scope tanımlanan değişken

What do we use let/const for?

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

const prevents you from making any assignments to a variable after the initial assignment. This way, if you define a PI constant and someone wants to change it, JS will throw an error.

2.2 Arrow Functions

We may argue that we write our arrow functions, or function codes, in a more mathematical language.

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

You'll believe you're building a math function when we describe how normal functions are defined using arrow functions below. You'll notice that this style of writing makes it easier to create code and improves code readability.

As instances of alternative Arrow function definitions, consider the following.

- No arguments constant function
- A pure function that takes a parameter and returns the answer
- A definition that does not need parameter encapsulation() since it only has one argument
- and so on...

```
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 improve code readability? If we use the Higher Order Function from above. The following is an example of a normal function.

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

You can see how easy it is to write the same code in Arrow Function. This is the ES6 Arrow function's power.

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

extras provided by the arrow function

The following ideas are a little difficult to comprehend. These concepts will be explained further in Deep JS topics. I just wanted to briefly mention them here.

Note 1: There is no constructor or prototype for arrow functions. It does not work with ***new***. Its aim is not to make an instance of an object.

Because **Note:2** arrow functions do not bind ***this binding***, the lexical scope context does so automatically.

2.3 Classes

Although the JS language is not an object-oriented language, OOP (Object Oriented Programming) can provide a language syntax and grammar that will give developers who are acclimated to it a comparable sense.

```
class User{
  constructor(name,age){
       this.name=name;
       this.age=age
    }

  sayHello(){
       console.log(`My name is ${this.name} and I am ${this.age}  years
old.`)
}}

const onur=new User("Onur",39);
onur.sayHello();
```

Now consider the following example. In the background, how does the class structure function?

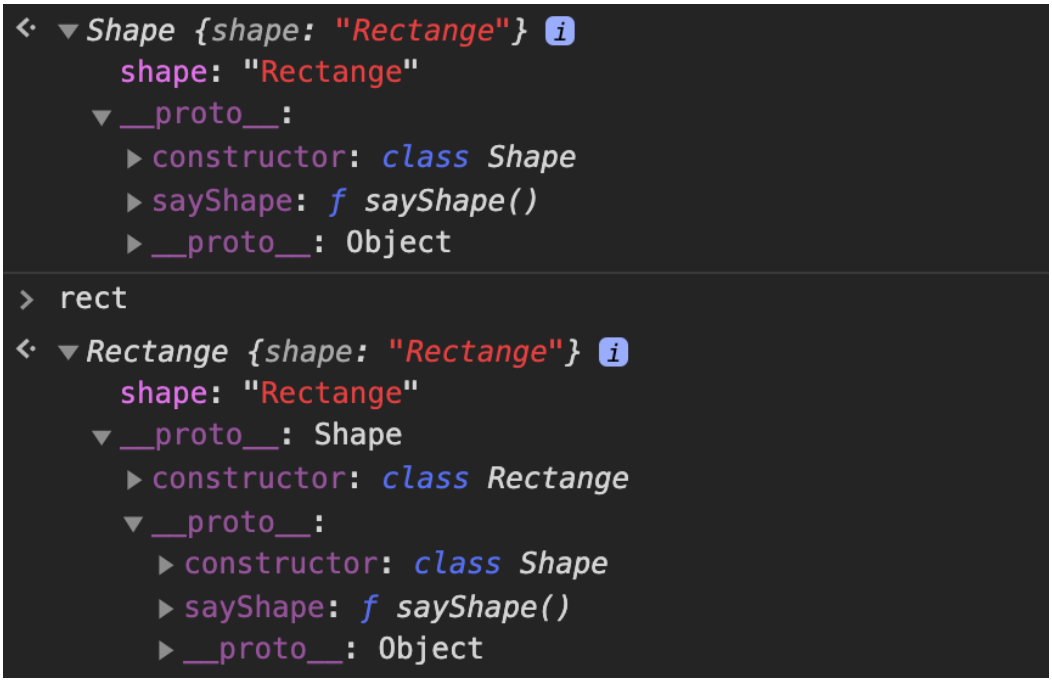
In reality, everything continues to function through the prototype. For example, see the two classes listed below.

- The former is **Shape,**
- whereas the later is derived from **Rectangle.**

```
class Shape {
  constructor(shape) {
    this.shape = shape;
  }

  sayShape() {
    console.log(`My Shape is ${this.shape}`);
  }
}

class Rectange extends Shape{
    constructor(){
      super("Rectange")
    }
}

const shape = new Shape("Rectange");
console.log(shape.sayShape());
const rect=new Rectange();
console.log(rect.sayShape());
```

Rectangle extends Shape

Despite the fact that Rectangle lacks a method, we may use its parent's sayShape method. So how?. The answer to the inquiry may be deduced from the data structure that has been stored in the background. You can see how the sayShape Rectangle is replicated in Shape under **__proto__**. By executing chain functions in the javascript language, this prototype may give inheritance. The key is right here.



2.4 Default Parameters

There are if statements that check if the JavaScript values are always undefined and apply default values to them. In such circumstances, using Default Parameters improves the look and readability of your code.

```
//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;
}
```

In certain circumstances, the variable is set a default value at the beginning and subsequently this value is created from the variables provided into it. For example, in the following addition operation, when we pass undefined values, we utilize 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
```

2.5 Template Literals

The new String definition method is => `backsticks` .

- **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...'
```

2.6 Destructing Assignments

Destruction is a notion that is used instead of the assignment/assign technique we are familiar with to make it easier to access variables in structures we generate (packed) in an object or array. This makes the code more understandable as well as shorter.

For simplicity and clarity, in the example below you can see 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.

Obje Deconstruction

```
const props={ name:'Hello', surname:'World'}
const 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

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

2.7 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 the other example, we can define the object very simply when we use [h] during object initialization in case the variable name is taken from outside.

```
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);
```

2.8 For-of Loop

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 increase 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(key)
for(let keyVal of Object.entries(me)) console.log(keyVal)
```

2.9 Promises

First of all, we were using Callbacks to handle async calls, these had some problematic parts and Promise structure was introduced to solve them. In short,

Callback

You can see how easy it is to create the CallbackHell structure. You can lose readability of the code in a very short time.

```
const waitThenCall = (callback) =>
                        { setTimeout(() => { callback() }, 1000)}

waitThenCall(() => {console.log("Say Hello1");
    waitThenCall(() => {console.log("Say Hello2");
       waitThenCall(() => { console.log("Say Hello3");})})})
```

Promise

With Promise structure

- We can get rid of the CallBack Hell structure through the then chain
- We also have the chance to handle **err** states.
- There is also a **catch block** to handle all process err state default.

```
const waitThenCall = (msg) => {
   returnnew Promise((resolve, reject) => {
         setTimeout(() => {resolve(msg + “success”) }, 1000)});}

waitThenCall(“Hello1_”)
.then((data) => { console.log(data)
                    return waitThenCall(“Hello2_”)},(err)=>{})
.then((data) => { console.log(data)
                    return waitThenCall(“Hello3_”)},undefined)
.then((data) => { console.log(data)})
.catch(err) => {console.log(err+"X")})
```

2.10 ES Modules

In JavaScript, over the years, we have been working on whether we can gather different module loading methods and specifications under a single standard. With **IIFE, CJS, AMD**, they thought about how to present module loading methods directly in JS Standards.

- JS içerisindeki **built-in** modül yapısı bu şekilde olmalı.
- **Tüm tarayıcılar** tarafından desteklenmeli
- **Asenkron(Async)** çalışmalı

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

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>
```

2.12 Spread operator

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

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

So this time you have array values in the background and you want to distribute them like values outside the array. 3 dots at the beginning of the object allows this to be distributed, you can use a similar structure in Object.

```
const obj1={a:1,b:2};
const obj2={...obj1};

obj1['c']=3;
console.log(obj2);
```

2.13 Set/Map

Object bize yetiyordu, Neden Map tipine ihtiyaç duyduk ?

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() =>constructor

get(key) => function that returns 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 the 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 the corresponding types as array.

for (let pair of mapObj) { //Loop
    var [key, value] = pair;
    console.log(key + " = " + value);
}
```

Array was enough for us, why did we need Set type?

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

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

But it's 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 again 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() =>constructor
has(value) => key değeri var mı? size => eleman sayısı
delete(value) => elemanı silerfor

(let value of setObj) { //Loop
    console.log(value);
}
```

Here in an important conversion **Arr → Set, Set → Arr** conversion must be very simple Below you can see how simple these 2 are done in a single line.

```
[…new Set(arr)]
```

2.14 Generators

Normally, functions terminate in 3 ways.

- completion of the operation of the function.
- return function returning the result
- Returning an error with throw

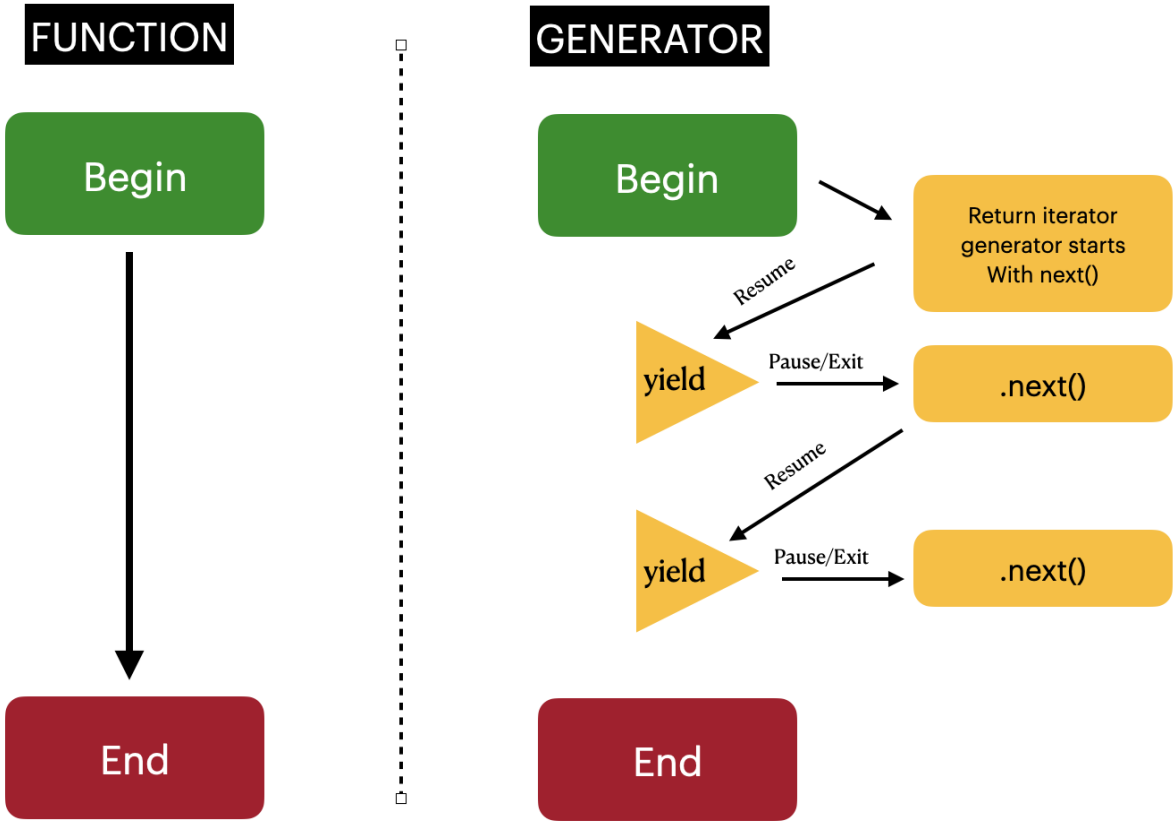
Generator treats the function like an iterator and turns it into a mechanism where you can stop and advance the flow within the function. What does this mean? When we call the **sayHello()** function in the function below, the entire function is executed immediately and the result is returned to you. You can say that async/await functions are not executed immediately. The function is held for async operation and this function type is already executed on the Generator structure.

```
function sayHello(){
  console.log("Hello");
  console.log("Word");
}

sayHello();
```

You can use the generator function by defining it with * in front of it and specifying the places where you will control the flow with **yield**, and you can use the flow of the function from the outside with **.next()**.

```
function * sayHelloGenerator(){
  yield console.log("Hello");
  yield console.log("Word");
}

const genHello=sayHelloGenerator();
genHello.next();
genHello.next();
```

Here the operation of 2 function types is shown. In the first one we cannot influence the function execution, while in the generator function we have the control to stop and advance the programme with the iterator.



Here yield behaves like return, but return returns not only value but also the result of the completion of the function.

```
function * sayHelloGenerator(){
  yield "Hello";
  yield "World";
}

const genHello=sayHelloGenerator();
console.log(genHello.next());
console.log(genHello.next());
console.log(genHello.next());
```

```
[object Object] {
  done: false,
  value: "Hello"
}
[object Object] {
  done: false,
  value: "World"
}
[object Object] {
  done: true,
  value: undefined
}
>
```

3.2016's JS language improvements (ES7)

3.1 Array.prototype.includes

Before the Array includes method, we used the Array **indexOf** method in most places and said that if the value is more than -1, it contains this value. But when we look at it, the purpose of the indexOf method is different. We don't want to know how many elements we want to understand whether the array contains this element

```
const arr=['red','green','blue'];
console.log(arr.includes('green'));
```

Aynı durumu **String** türünde de görebiliriz.

```
str.includes('substring')  gives true/false
str.indexOf('substring') !== -1  gives position check -1
str.match(/substring/g)> 0 gives count or repeat
(new RegExp('substring')).test(str)
```

3.2 Exponentiation operator (**)

To say 2 over 4, you can use ** commands instead of Math.pow.

```
console.log(Math.pow(2,4)); //16
console.log(2**4); //16
```

3.3 Array.prototype.find and Array.prototype.findIndex

find() returns the element in the Array that matches the condition

findIndex() Returns the position, i.e. index, of the element in the Array that matches the condition

```
const arr = [{name: 'Ahmet', age: 30},
             {name: 'Ali', age: 25},
             {name: 'Veli', age: 35}];

console.log(arr.find(person => person.age === 25));
// {name: 'Ali', age: 25}


console.log(arr.findIndex(person => person.age === 25));
// 1
```

3.4 Object.getOwnPropertyDescriptors

The properties of the object's property value give information about **value, writable, configurable, enumerable, get, set**.

```
const me={ name:"Onur", age:39, height:180}
console.log(Object.getOwnPropertyDescriptor(me,'age'));
//{value:39,writable:true,enumerable:true,configurable:true}
```

4. 2017's JS language improvements (ES8)

4.1 String padding

String allows you to complete up to a certain character.

- whether to complete with a blank character / with a certain pattern.
- **padStart/padEnd:** whether to start/end the fill operation

```
console.log('abcd'.padStart(4));  //'abcd'
console.log('abcd'.padStart(8));  //'    abcd'
console.log('abcd'.padStart(12)); //'        abcd'
console.log('abcd'.padStart(12,'xyz'));

//'xyzxyzxyabcd'console.log('abcd'.padEnd(12)); //'abcd        '

console.log('abcd'.padEnd(8));  //'abcd    '
console.log('abcd'.padEnd(4));  //'abcd'
console.log('abcd'.padStart(12,'xy')); 'xyxyxyxyabcd'
```

4.2 Object (values, entries)

values(): Returns the object's own values in the object as an array.

```
const me={ name:"Onur", age:39, height:180}
console.log(Object.values(me)); //["Onur",39,180]

const names=['Onur','Ali','Veli'];
console.log(Object.values(names)); //['Onur','Ali','Veli']
```

entries(): Returns the object's own values [key,value] as an array.

```
const me={ name:"Onur", age:39, height:180}
console.log(Object.entries(me)); //[{"name":"Onur"},...]

const names=['Onur','Ali','Veli'];
console.log(Object.entries(names)); //[{0:'Onur'},...]
```

4.3 Async Functions

JS provides a number of methods to control the output of async functions, callback, promise and finally async functions

Callback

You can see how easy it is to create the CallbackHell structure. You can lose readability of the code in a very short time.

```
const waitThenCall = (callback) =>
                        { setTimeout(() => { callback() }, 1000)}

waitThenCall(() => {console.log("Say Hello1");
    waitThenCall(() => {console.log("Say Hello2");
       waitThenCall(() => { console.log("Say Hello3");})})})
```

Promise

With Promise structure

- We can get rid of the CallBack Hell structure through the then chain
- We have a chance to handle **err** states.
- There is also a **catch block** for all process err state default handle.

```
const waitThenCall = (msg) => {
   returnnew Promise((resolve, reject) => {
         setTimeout(() => {resolve(msg + “success”) }, 1000)});}

waitThenCall(“Hello1_”)
.then((data) => { console.log(data)
                    return waitThenCall(“Hello2_”)},(err)=>{})
.then((data) => { console.log(data)
                    return waitThenCall(“Hello3_”)},undefined)
.then((data) => { console.log(data)})
.catch(err) => {console.log(err+"X")})
```

Async/Await

Actually, this is a topic I will talk about in the future, but in summary, with the async/await that comes with ES8, you can write your code like a normal synchronous code instead of this then. This time we put a tag in the code. Look, this function is async, this call is asynchronous, wait here with await. You can also use the **try - catch** mechanism directly.

```
const waitThenCall = (msg) => {
    return new Promise((resolve, reject) => {
       setTimeout(() => { resolve(msg + "success") }, 2000)});}

  async function run(){

    try{
     const result=await waitThenCall("Hello1");
     console.log(result)
     const result2=await waitThenCall("Hello2");
     console.log(result2)
     console.log("Hello")
    }
    catch(e){
     console.log(e);
    }

 run();
//Result
//Hello1success
//Hello2success
//Hello
```

4.4 Shared Memory And Atomics

JavaScript, **shared memory and atomics,** are capabilities developed to run multi-threaded programming more effectively and efficiently.

Shared memory allows multiple threads to access the same memory region, allowing faster and more efficient communication between threads.

The **SharedArrayBuffer** object is used to create a shared memory buffer that can be accessed by multiple threads.

Atomics provides a way to perform low-level atomic operations on shared memory locations. These operations are guaranteed to be indivisible and cannot be interrupted by other threads, ensuring that the shared memory is accessed safely and correctly.

```
// Create a shared memory buffer
const buffer = new SharedArrayBuffer(4);

// Create an Int32Array view of the shared memory buffer
const view = new Int32Array(buffer);

// Create two worker threads to increment the shared memory location
const worker1 = new Worker('worker.js');
const worker2 = new Worker('worker.js');

// Send the shared memory buffer to the worker threads
worker1.postMessage(buffer, [buffer]);
worker2.postMessage(buffer, [buffer]);

// In the worker.js file:
self.onmessage = (event) => {
  const buffer = event.data;
  const view = new Int32Array(buffer);

  // Increment the shared memory location atomically
  Atomics.add(view, 0, 1);
}
```

5. 2018's JS language improvements (ES9)

5.1 Async Iteration

It is the ability of **for-await-of loop**, which can provide iterate on collections asyncly.

```
const asyncIterable = {
  [Symbol.asyncIterator]() {
    const data = ['hello', 'world'];
    let index = 0;
    return {
      next() {
        if (index < data.length) {
          return Promise.resolve({ value: data[index++], done: false
});
        } else {
          return Promise.resolve({ done: true });
        }
      }
    };
  }
};

(async function() {
  for await (const item of asyncIterable) {
    console.log(item);
  }
})();
```

5.2 Rest/Spread

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

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

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

Let's look at the same in 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 it gives the possibility to pass 0 or more parameters as an array as varargs(...) in Java language.

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

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

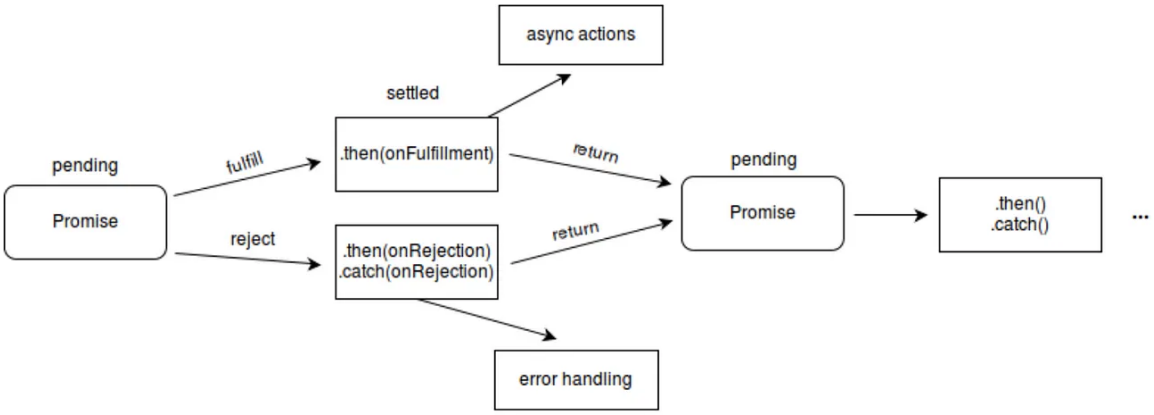
```
function sum(numbers){
    let toplam=0; numbers.forEach(el=>toplam+=el);
    return toplam;
}
console.log(sum([2,4,6,5])); //17
```

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 caller part.

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

5.3 Promise.prototype.finally

Promise **prototype dependent (then, catch, finally)** methods **then()** acts as a conjunction that allows you to create promise chains. catch() allows you to catch errors that were not handled during reject. **finally()** This method is called when the promise finally completes successfully or unsuccessfully.




```
const promise = doSomething();
promise.then(result => {
  console.log(result);
}).catch(error => {
  console.error(error);
}).finally(() => {
  console.log('Done');
});
```

5.4 RegExp Improvements

- Unicode property escapes,
- named capture groups,
- lookbehind assertions.

6. 2019's JS language improvements (ES10)

6.1 Array (flat, flatMap)

flat() converts multidimensional array structures to one-dimensional.

```
const a0=['AA', ['BB', 'CC']].flat(); console.log(a0);
//['AA', 'BB','CC']
const a1=['AA', ['BB', ['CC']]].flat(2); console.log(a1);
//['AA', 'BB','CC']
const a2=['AA', ['BB', ['CC', ['DD']]]].flat(Infinity);
console.log(a2); //['AA', 'BB','CC']
```

flatMap() ise map donen array değerini tek boyutlu bir hale getirir.

```
const b0=['Aa bb', 'cc dd'].map(words => words.split(' '));
console.log(b0); [[AA bb],[cc dd]]

const b1=['Aa bb', 'cc dd'].flatMap(words => words.split(' '));
console.log(b1); [AA, bb, cc, dd]
```

6.2 Optional catch binding

Allows you to skip the identifier identifier in the catch block in case you don't need a caught error.

```
try {
  // some code that may throw an error
} catch (error) {
  // handle the error (with the 'error' binding)
} catch {
  // handle the error (without referencing it)
}
```

6.3 Object (fromEntries)

It is used to create an object clone via Entries.

```
const me={ name:"Onur", age:39, height:180}

const entries=Object.entries(me);
console.log(entries); //[{"name":"Onur"},...]

const meClone=Object.fromEntries(entries)
console.log(meClone)
```

6.4 String (trimStart, trimEnd)

If there are spaces at the beginning or end of a string, delete them.

```
' AAAA'.trimStart() //'AAAA'
' AAAA '.trimStart() //'AAAA '
' AAAA '.trimEnd() //' AAAA'
'AAAA '.trimEnd() //'AAAA'
```

6.5 Symbol (description)

In order to better follow and debug the code in the symbol, it has allowed to enter a description.

```
const mySymbol = Symbol('This is a symbol with a description');
console.log(mySymbol.description); // 'This is a symbol with a
description'
```

6.6 Array.prototype.sort() - Stable Sorting

Array structure sort operation was not guaranteed to be stable before ES10 and this difference could vary depending on the JavaScript engine used.

The sort() operation, which now receives a comparison function, performs the sorting operations according to the negative and positive values returned from here. And the sorting process is successful.

```
const arr = [  { id: 2, name: 'Alice' },  { id: 1, name: 'Bob' },  {
id: 3, name: 'Charlie' },  { id: 4, name: 'Bob' }];

arr.sort((a, b) => {
  if (a.name < b.name) {
    return -1;
  } else if (a.name > b.name) {
    return 1;
  } else {
    return a.id - b.id;
  }
});

console.log(arr);
// Output: [
//   { id: 1, name: 'Bob' },
//   { id: 4, name: 'Bob' },
//   { id: 2, name: 'Alice' },
//   { id: 3, name: 'Charlie' }
// ]
```

6.7 Well-formed JSON (stringify)

In previous versions of ECMAScript, the JSON.stringify() method serialised certain data types in ways that did not conform to the JSON specification. For example, NaN, Infinity and -Infinity values were serialised as null and undefined values were omitted entirely.

With the introduction of the "Well-formed JSON.stringify" feature in ES10, the JSON.stringify() method has been updated to produce JSON output that is more consistent with the JSON specification. This means that NaN, Infinity and -Infinity values are now serialised as "null" and undefined values are serialised as "undefined".

```
const obj = { a: NaN, b: Infinity, c: -Infinity, d: undefined };
const jsonString = JSON.stringify(obj, null, 2);
console.log(jsonString);
// Output: {"a":null,"b":null,"c":null,"d":"undefined"}
```

6.8 Static Field

The features found in the Java language are slowly coming into the JavaScript language. One of them allows us to create functions directly dependent on the Class definition without creating a Class with a function defined with static

with new.

```
class Smartphone {
  add_color() {
    console.log("Adding Colors");
  }
}
const apple = new Smartphone();
apple.add_color(); // output is: Adding Colors

Smartphone.add_color() // TypeError: Smartphone.add_color is not a function
```

Static Fields.js hosted with ❤ by GitHub view raw

```
class Smartphone {
  designer(color) {
    this.color = color;
  }
  static create_smartphone(color) {
    return new Smartphone(color);
  }
}
const silver = Smartphone.create_smartphone("silver"); // output is: undefined
```

Static Fields 2.js hosted with ❤ by GitHub view raw

7. 2020's JS language improvements (ES11)

7.1 BigInt

JavaScript 2^{53} - 1 is the maximum number you can represent in JavaScript. `Number.MAX_SAFE_INTEGER` is used as the value. This makes the value 9007199254740991. When you do operations on Integer, when you add 2, when you add 4, you can see that it now gives incorrect values. It is no longer possible to perform reliable operations on this number.

Now you can safely perform your operations with large numbers by adding the n character to the end of the numbers or using the number environment via **BigInt(number)**.

Integer

```
> var maxIntNum = 9007199254740991;
< undefined
> console.log(maxIntNum)
  9007199254740991
< undefined
> console.log(maxIntNum+1)
  9007199254740992
< undefined
> console.log(maxIntNum+2)
  9007199254740992
< undefined
> console.log(maxIntNum+3)
  9007199254740994
< undefined
> console.log(maxIntNum+4)
  9007199254740996
< undefined
>
```

BigInt

```
> const maxNum = 9007199254740991n;
< undefined
> console.log(maxNum)
  9007199254740991n
< undefined
> console.log(maxNum+1)
✖ Uncaught TypeError: Cannot mix BigInt and other types, use explicit conversions
    at <anonymous>:1:19
> console.log(maxNum+1n)
  9007199254740992n
< undefined
> console.log(maxNum+2n)
  9007199254740993n
< undefined
> console.log(maxNum+3n)
  9007199254740994n
< undefined
> console.log(maxNum+4n)
  9007199254740995n
< undefined
>
```

7.2 Dynamic import

As a result, in Modern JavaScript, you could do **import** operations statically at the top of the JavaScript package.

```
import ... from  'msg'   // diyebilirsiniz fakat module en başında
statik
```

In fact, the above static import is very useful in terms of making the code more reliable and less fragile.

But on the Web, we need a lot of code not from the beginning of the application, but as we use it.

In this case, it may be necessary to include some code in your application later, both in terms of a flexible architecture and performance.

```
//Aşağıdaki şekilde kullanıma izin verilmez

if() import ... from  'msg' //Condition
{import ... from  'msg'} //Blok icerisinde
```

In other words, it does not allow us to import a button dependent, a condition dependent import in our code. But with the Dynamic Import method, it allows you to import and use the Runtime module.

```
import("/msg").then(({hello, world}) => {
    hello();
    world();
});

//Veya..

let {hello, world} = await import('./msg');
hello();
world();
```

7.3 Nullish Coalescing

In Normal, when we had a **falsy condition**, it gave us the other side of the OR operator.

```
const format = (value) => {
  const defaulted = value || 'default';
  return `I formatted ${defaulted} for you.`;
};
```

A new operator, **the ?? sign (false, 0 "")**, which will handle the other part only in the case of undefined and null, but not in all falsy cases, is that it only works in undefined and null cases.

```
const format = (value) => {
  const defaulted = value ?? 'default';
  return `I formatted ${defaulted} for you.`;
};
```

Because concepts such as 0, 0n, false, "" on the left-side may be meaningful in our application. In this case you can use ?

7.4 Optional Chaining

JS developers often encounter this type of "**TypeError: Cannot read property "** error is frequently encountered. The absence of an object in an object during property access causes this type of error. Here, firstly that object must be null checked and such evaluations must be made in an if.

This creates the need for extra if checks in many parts of the code. (Note: The emergence of languages such as TypeScript and protecting developers from extra if checks is also a method. But JS has introduced the Optional

Chaining feature that makes its job easier.

As can be seen in the example below, we get a TypeError when we want to access a property of a non-existent object.

```
const person= {
    bag: {
        orange: "2kg"
        banana: "1kg"
    }

}

console.log(person.bag.orange) // output is: ‘2k’
console.log(person.bag.patato) // output is: 'undefined'

console.log(person.wallet.coin)
// TypeError: Cannot read property 'coin' of undefined
```

Optional Chaining has introduced a method to prevent this.

```
console.log(person.wallet?.coin) // output is: undefined
```

We can use the same controlled call in array and function calls.

```
console.log(userList?.[1]) //Array item access
console.log(getWords?.())  //Function acess
```

7.5 Promise.allSettled

The allSettled method has been added to Promise methods. It returns a detailed result about the fulfilled and unfulfilled promise.

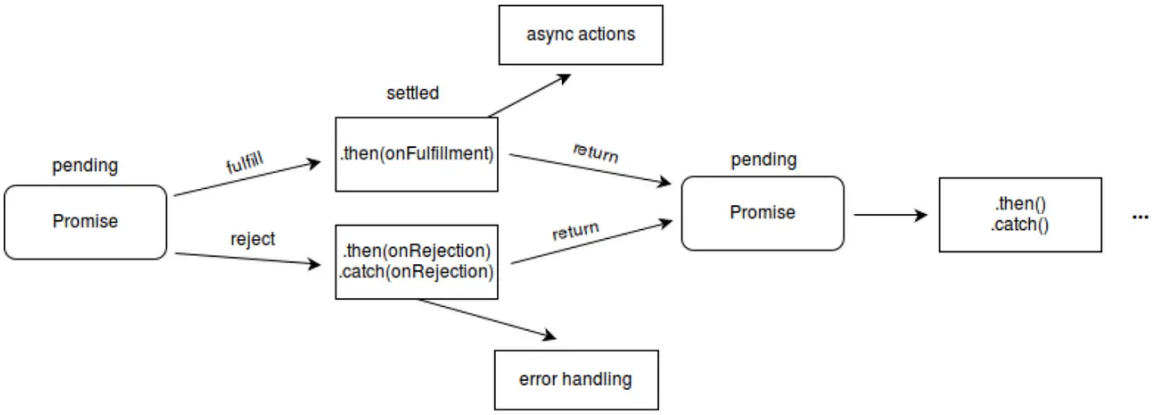
Promise Methods

In promise **prototype bound (then, catch, finally)** methods, **then()** acts as a conjunction that allows you to create promise chains. **catch()** allows you to catch errors that were not handled during reject. **finally()** This method is called when the promise finally completes successfully or unsuccessfully.

The **(resolve, rejected)** methods are called after the successful/unsuccessful completion of the promise. Like Promise.resolve(value), Promise.reject(err).

Methods that manage multiple Promise conditions **(all, race, allSettled)**

- **all:** Waits for all Promises to be successfully completed.
- **race:** Gets the result of whichever Promise is completed first.
- **allSettled:** When all Promise successful and unsuccessful operations are finished, it returns the results with their status.



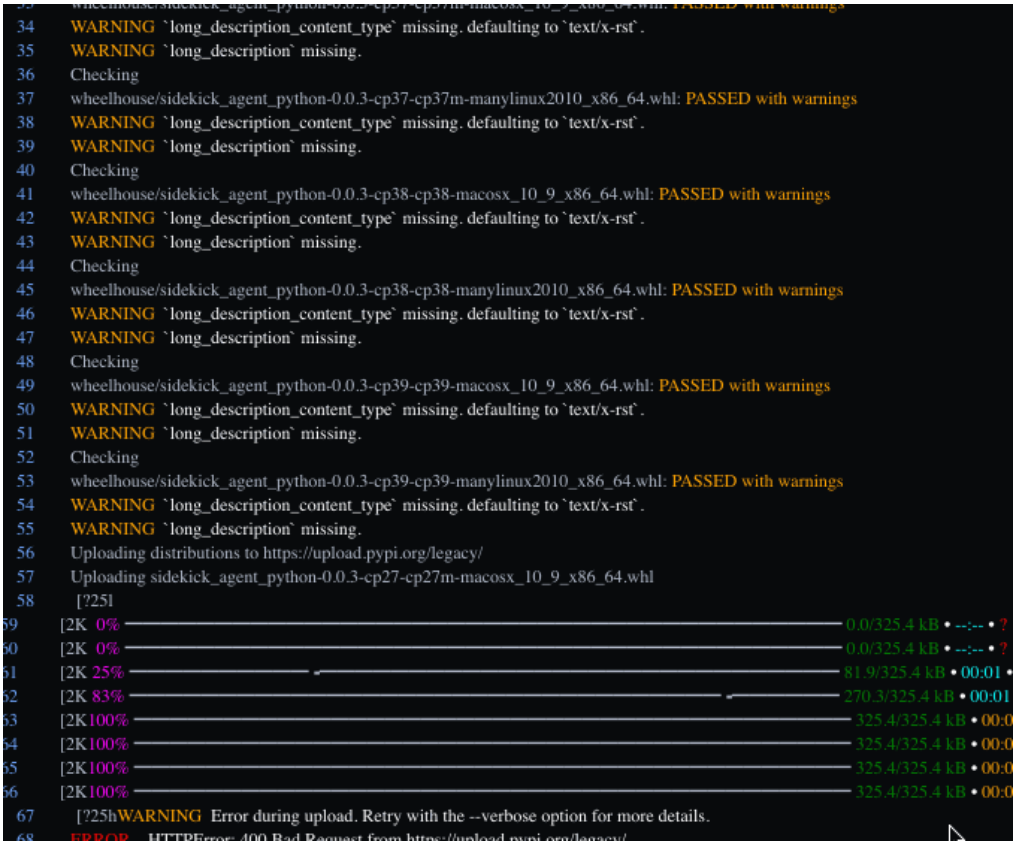
7.6 String#matchAll

When using Regular Expression, the word match finds all the words in which this regex occurs, while the related indexes, etc... does not find them, instead it just returns the word. When you need to perform an operation over the indexes of these words, you need to run separate logics. For example, in the parts we want to colour in GitHub LogViewer (Example application)

Colour codes had to be found and changed.

```
calcColoring = logLine => {
    const matches = logLine.match(/\x1b\[\d+m/g);
    if (hasArrayElement(matches)) {
      return logLine
        .replace(/\x1b\[\d+m/g, '$!$')
        .split('$!$')
        .map((el, index) => {
          const styleCode = matches[index - 1];
          return (
            <span key={index} style={logLineStyle[styleCode]}>
              {el}
            </span>
          );
        });
    } else {
      return logLine;
    }
  };
```

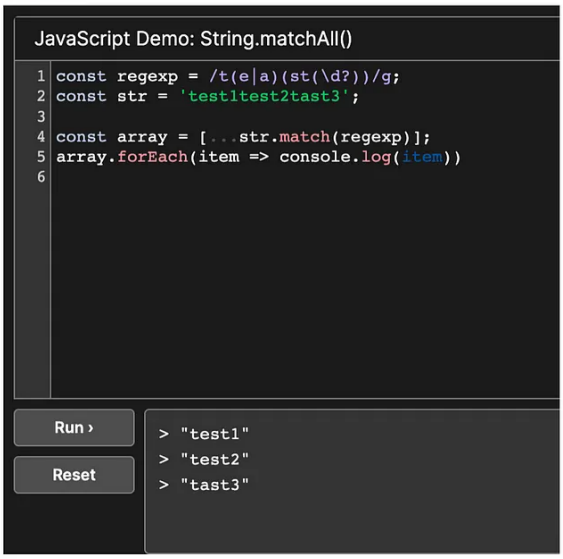
Yukarıda ben bu indekslere erişim için ayrıca $!$ kodları yerleştirdim..



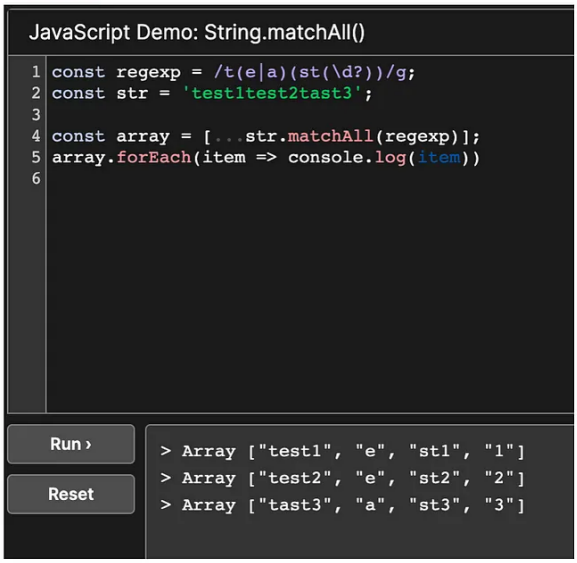
Since we do not know which of these colour matches which match, I had to turn to such replace methods.

MatchAll, on the other hand, allows us to operate our application logic in a better way by returning the reasons for each match with each part in the array.

Match

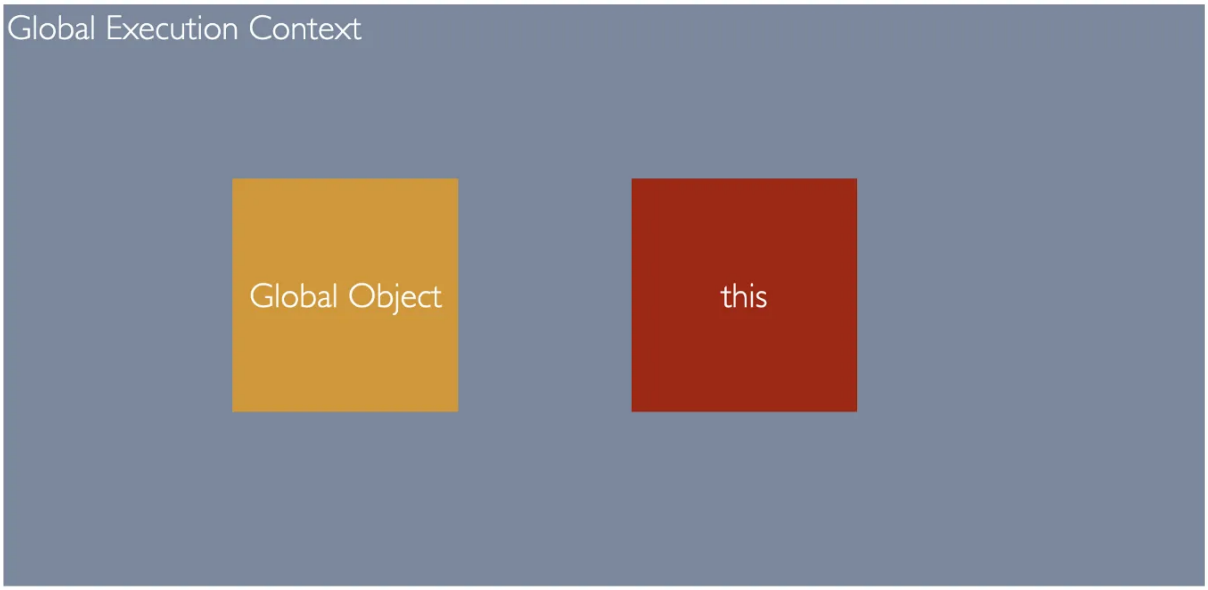


MatchAll

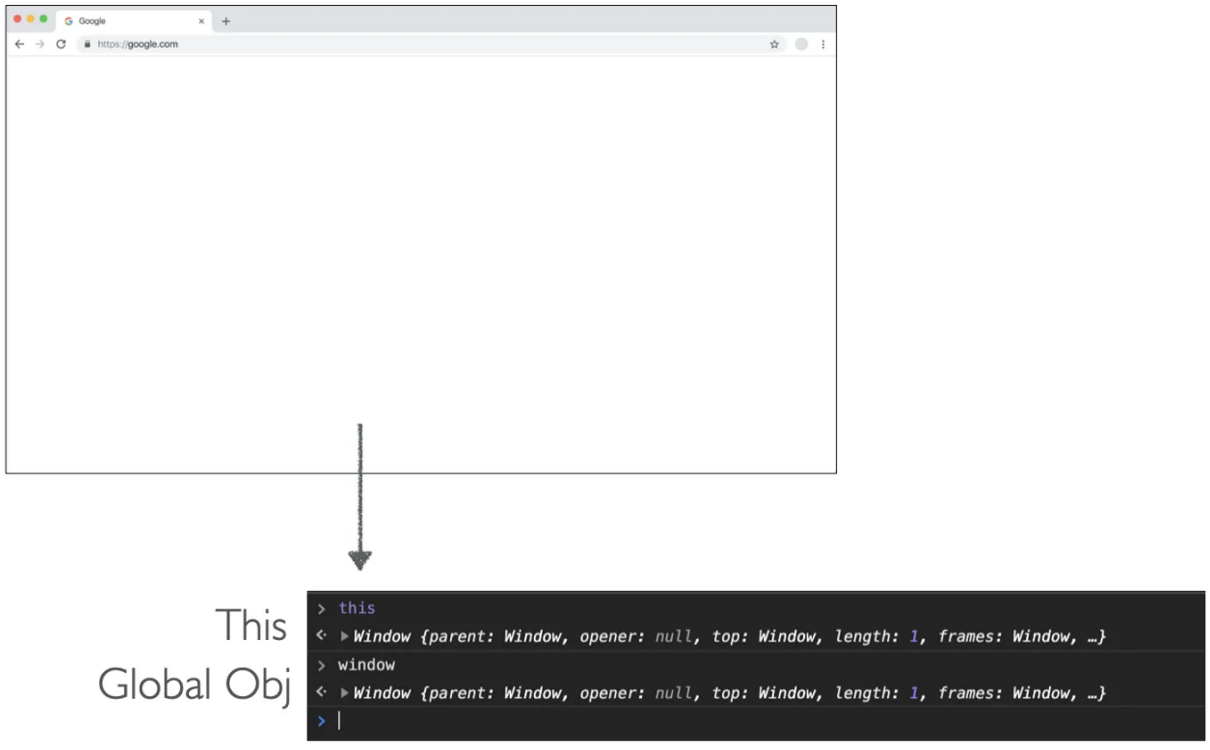


7.7 globalThis

Whether JS is running in a browser, server or local, JS Engine creates a Global Execution Context in the working environment and provides access to WebAPI for browsers from this Global Scope. This first Global Execution Context contains a **Global Object** and this object. JSEngine places this object when it first stands up in the JS execution environment.



For example, Console this and window correspond to the Global Object for JS codes running in the Browser below.



In the browser and NodeJS Runtime environments, the globalThis variable has been added so that the Global Context can access different Runtimes in the same way.

```
// In a browser
window == globalThis // true

// In node.js
global == globalThis // true
```

7.8 Module Namespace Exports

Allows a module to define a single object that serves as a namespace for all its exports. This feature makes the code more concise and readable when making multiple exports from a module.

It is created using **export * as**. For example:

```
// module.js
export const foo = 'foo';
export const bar = 'bar';
export const baz = 'baz';
export * as names from './names.js';
```

7.9 Well defined for-in order

You usually use **for...in** when travelling the elements of the object. Since the retention of these elements was not in an array but like a map, there was no guarantee that the order would be during the definition in the Object. With this feature, it is ensured that the elements from this loop give the Object elements in a certain order.

```
const obj = { c: 1, a: 2, b: 3 };

for (const prop in obj) {
  console.log(prop);
}

//c, a, b,
```

7.10 import.meta

Dynamic import additionally provides a meta attribute containing the information of the currently imported module. Currently there is an url attribute inside, representing the URL that the module references.

7.11 private fields (#)

Before 2015, it was a problem that these variables (var) functions could be accessed from everywhere. **IIFE functions** were used for this

```
// Define "a" in global scope
var a = 123;

// Define "b" in function scope
(function() {
  console.log(b); //=> Returns "undefined" instead of an error due to
hoisting.
  var b = 456;
})();

console.log(a); // => 123
console.log(b); // Throws "Ref
```

If a function is called immediately and the result/result object is stored in a variable, this type of function is 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, in libraries such as jQuery, Prototype, the IIFE method is used to avoid conflicting variable names and to perform operations within the scope.

Let, Const

With ES6 variables variables, we started to define variables at blog level with **let, const**. We have become able to create private and public property structures with function module pattern structure.

Function Module

```
const CarModule = () => {
  let milesDriven = 0;
  let speed = 0;

  const accelerate = (amount) => {
    speed += amount;
    milesDriven += speed;
  }

  const getMilesDriven = () => milesDriven;

  // Using the "return" keyword, you can control what gets
  // exposed and what gets hidden. In this case, we expose
  // only the accelerate() and getMilesDriven() function.
  return {
    accelerate,
    getMilesDriven
  }
};

const testCarModule = CarModule();
testCarModule.accelerate(5);
testCarModule.accelerate(4);
console.log(testCarModule.getMilesDriven());
```

```
function CarModule() {
  let milesDriven = 0;
  let speed = 0;

  // In this case, we instead use the "this" keyword,
  // which refers to CarModule
  this.accelerate = (amount) => {
    speed += amount;
    milesDriven += speed;
  }

  this.getMilesDriven = () => milesDriven;
}

const testCarModule = new CarModule();
testCarModule.accelerate(5);
testCarModule.accelerate(4);
console.log(testCarModule.getMilesDriven());
```

ES6 Classes

With ES6 Classes, we have been able to create variables that you can create your variables in the Constructor and use them in the functions of the class, but these still did not completely remove accessibility from the outside. All private variables were defined with _ in order to be clear, of course, this structure is not an obstacle to be changed from the outside, the developer was just a variable name definition to increase visibility to inform private variables.

Or, in order to provide full privacy, it is necessary to define all variables and functions in the constructor method, which is not a correct approach in Class usage.

Private Fields

Thanks to the new private field capability, it is possible to define variables that can only be accessed from within the class.

```
class User {
  #name = "Onur";

  getName() {
    return this.#name;
  }
}
const user1= new User();
console.log(user1.getName()); // Onur
console.log(user.#name) // Private field error
```

8. 2021's JS language improvements (ES12)

8.1 Numeric Separators

A feature to make numeric values more readable with _ underscore

For example, let's examine the numbers below. The figures I wrote after comment on the right side are more readable.

```
const a=10000000000000; // 1,000,000,000,000
const b=1019436871.42  // 1,019,436,871.42
```

,. usage may differ according to languages. Since we only want to format the numbers here, the following representation makes it easier to read the numbers in the code.

```
1_000_000_000_000
1_019_436_871.42
```

In addition to all these, you can also use these brackets in different number definitions.

```
let binary = 0b1010_0001_1000_0101; //
let hex= 0xA0_B0_C0;
let bigInt=  9_223_372_036_854_775_807n
```

8.2 String.prototype.replaceAll

We have added replaceAll to the String prototype ourselves for years.

```
String.prototype.replaceAll = function (search, replacement) {
  var target = this;
  return target.split(search).join(replacement);
};
```

now this code will come as JS default. In this way, it offers the opportunity to find and change all the words you want to change in the code.

```
> const input="This is a test program which contains test code";
< undefined
> console.log(input.replaceAll('test',"jest"))
< undefined
```

and result

```
This is a jest program which contains jest code
```

8.3 Promise.any() and AggregateError

When I try to process more than one promise together, at least one of them is the Promise call that returns resolve according to the resolve status.

```
const promises = [
  new Promise((resolve, reject) => setTimeout(reject, 20, '1st')),
  new Promise((resolve, reject) => setTimeout(resolve, 500, 'last')),
  new Promise((resolve,reject) => setTimeout(resolve, 200, '2nd'))
];

Promise.any(promises)
    .then(v => console.log(v))
    .catch(e =>console.log(e));
```

AggregateError returns multiple errors in an array as AggregateError in case of error as a result of Promise.any.

8.4 Logical Assignment Operators

In this section, when assigning a value, we make an if control beforehand and assign according to the state of the value.

- Logical And Assignment
- Logical Or Assignment
- Nullish coalescing operator.

Logical And Assignment (&&=)

In Logical And Assignment, if the x value is already a true value, it assigns it. Otherwise it does not assign.

```
//x &&= y;
//if x is true, then the value of y is assigned to x

//Example
let x =10, y =20;
x &&= y;
console.log(x); // 20

x = undefined;
x &&= y;
console.log(x); // undefined
```

Logical Or Assignment (||=)

In Logical Or Assignment, if the x value is already a false value, it assigns it. Otherwise it does not assign.

```
//x ||= y;
//if x is false, then the value of y is assigned to x

//Example
let x ='', y ="x is empty";
x ||= y;
console.log(x); // x is empty

x = 'asdf';
x ||= y;
console.log(x); //asdf
```

Nullish coalescing operator (??=)

Logical Nullish coalescing performs an assignment only in case of null and undefined.

```
let a ;
a = a ?? 10;
console.log(a)

// using shorthand assignment
let x;
x ??= 100;
console.log(x); //100

// only for null and undefined the right-side operand is assigned.
let emptyStr = '';
emptyStr ??= "Empty";
console.log(emptyStr); //''
```

8.5 Private Class Methods and Accessors

In ES11, the private ability of Class property values has become applicable to methods in the same way. In the following code, getToken can only be used from methods in the class because it is defined as a private method.

```
class Auth {
  #getToken() {
   return "12345678";
  }
  isAuth() {
   return this.#getToken();
  }
}
```

It also allows to define getter and setter functions.

9. 2022's JS language improvements (ES13)

9.1 .at() function for Indexing

We can access types such as String, Array via item or char **[index]** in the related index. You can now use the .at() function for this.

```
const arr = [100, 200, 300, 400];
arr[0]; arr.at(0); // 100
arr[-2]; arr.at(-2); // 300

const str = "ABCD";
str[-1]; str.at(-1); // 'D'
str[0]; str.at(0); // 'A'
```

9.2. Array find from last (findLast)

When searching in Array, find(), findIndex() were available. In order to search in reverse, the findLast() and findLastIndex() functions have been added to the Array prototype for the structure you need to search after saying String.reverse.

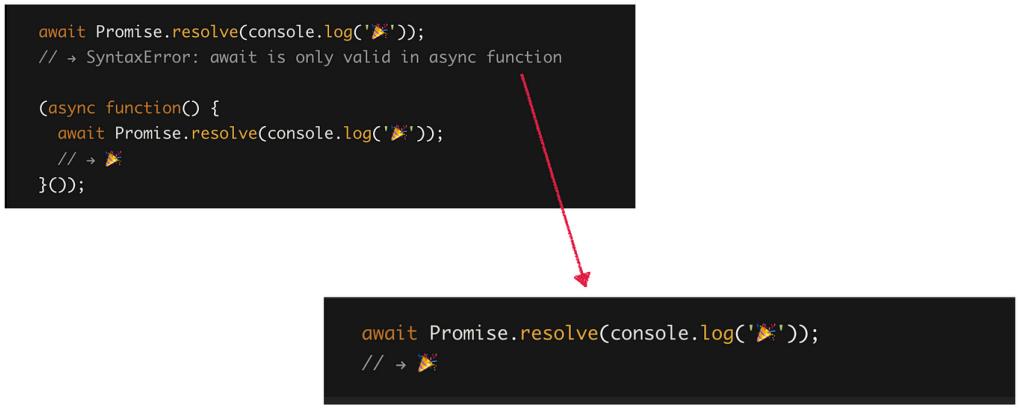
9.3. Error Cause (cause)

To help in the case of unexpected behaviour, errors need to be augmented with contextual information such as error messages and error instance properties to explain what happened at the time. The cause property on the Error object allows us to determine which error caused the other error.

```
try {
  apiCall()
} catch (err) {
  throw new Error("New error message", { cause: err });
}
```

9.4. Await operator at the top-level (await)

Previously the await keyword could only be used in async blocks,



You can now use it in any way you want from the top level outside this context. There are sample uses about this below.

```
//Dynamic Module Loading.
const strings = await import(`./example.mjs`);

//Fallback module Loading
let jQuery;
try {
  jQuery = await import("https://cdn-a.com/jQuery");
} catch {
  jQuery = await import("https://cdn-b.com/jQuery");
}

//Load module parallel...
const resource = await Promise.any([
  fetch("http://example1.com"),
  fetch("http://example2.com"),
]);
```

9.5. Class Field Declaration (#)

There is a similar section in the ES11 section, the same one, the part we describe as Private Fields. You can read it again here.

9.6. Ergonomic brand checks for Private Fields (in)

In order to check whether the private fields belong to the relevant object, it checks whether it is in that object with the in keyword. In the picture below, it allows the controls of the relevant parts to be done through the private prop.

https://plainenglish.io/blog/latest-es13-javascript-features

9.7. Class Static Block

It allowed to define static functions and variables in the class. Now it allows to define static block etc. as you want.

```
class Dictionary {
  static words = ["yes", "no", "maybe"];
}

class Words extends Dictionary {
  static englishWords = [];
  static #localWord = "ok";
  // first static block
  static {
    // link by super to the  Dictionary class
    const words = super.words;
    this.englishWords.push(...words);
  }

  // second static block
  static {
    this.englishWords.push(this.#localWord);
  }
}

console.log(Words.englishWords);
//Output -> ["yes", "no", "maybe", "ok"]
```

https://plainenglish.io/blog/latest-es13-javascript-features

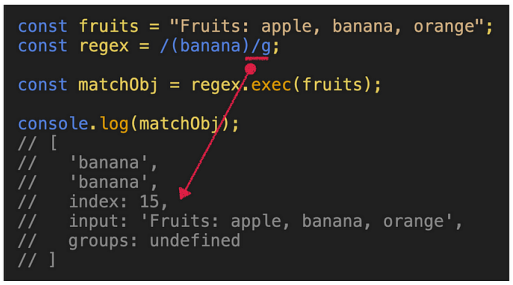
9.8.hasOwn

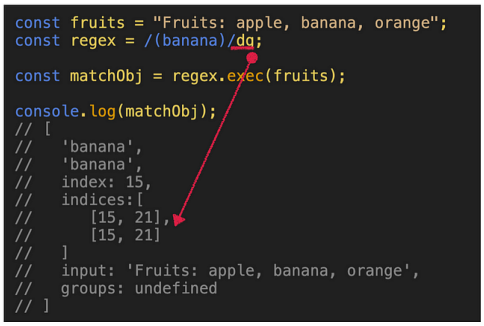
Object.prototype.hasOwnProperty function existed, but it was recommended to use it via prototype. Instead, the .hasOwn function that can be used directly from Object was developed.

```
const object = { name: "Mark" };
Object.hasOwn(object, "name"); // true

const object2 = Object.create({ name: "Roman" });
Object.hasOwn(object2, "name"); // false
Object.hasOwn(object2.__proto__, "name"); // true

const object3 = Object.create(null);
Object.hasOwn(object3, "name"); // false
```

9.9 RegExp Match Indices

It is now possible to access extra index data while working only on words via RegExp match. Below you can see the extra indexes when we call /g /dg.


const fruits = "Fruits: apple, banana, orange";
const regex = /(banana)/g;
const matchObj = regex.exec(fruits);
console.log(matchObj);
// [
//   'banana',
//   'banana',
//   index: 15,
//   input: 'Fruits: apple, banana, orange',
//   groups: undefined
//]


const fruits = "Fruits: apple, banana, orange";
const regex = /(banana)/dg;
const matchObj = regex.exec(fruits);
console.log(matchObj);
// [
//   'banana',
//   'banana',
//   index: 15,
//   indices:[
//   [15, 21],
//   [15, 21]
//   ]
//   input: 'Fruits: apple, banana, orange',
//   groups: undefined
//]