Table of Contents

1. Introduction

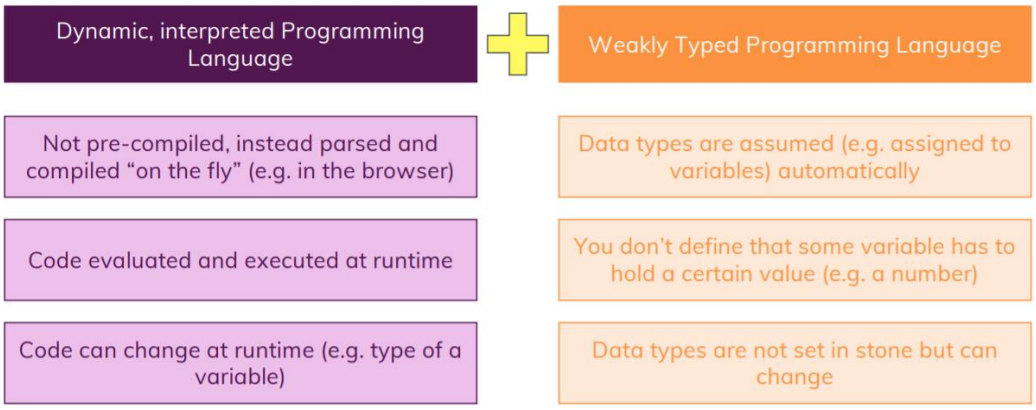
1.1 What is JavaScript?

> JavaScript is a **dynamic**, **weakly typed** programming language which is **compiled at runtime**. It can be executed as part of a webpage in a browser or directly on any machine (**"host environment"**).

> JavaScript was created **to make webpages more dynamic** (e.g. change content on a page directly from inside the browser). Originally, it was called LiveScript but due to the popularity of Java, it was renamed to JavaScript.

JavaScript is an interpreted language. The browser interprets the JavaScript code embedded inside the web page, executes it, and displays the output. It is not compiled to any other form to be executed.

All the modern web browsers come along with the JavaScript Engine, this engine interprets the JavaScript code. There is absolutely no need to include any file or import any package inside the browser for JavaScript interpretation.



1.2 Where to write JavaScript code?

JavaScript code can be embedded within the HTML page or can be written in an external file.

1.2.1 Internal Scripting

Internal scripting, is done with the help of HTML tag : <script> </script>
This tag can be placed either in the head tag or body tag within the HTML file.

1.2.2 External Scripting

JavaScript code can be written in an external file also. The file containing JavaScript code is saved with the extension *.js (e.g. fileName.js)

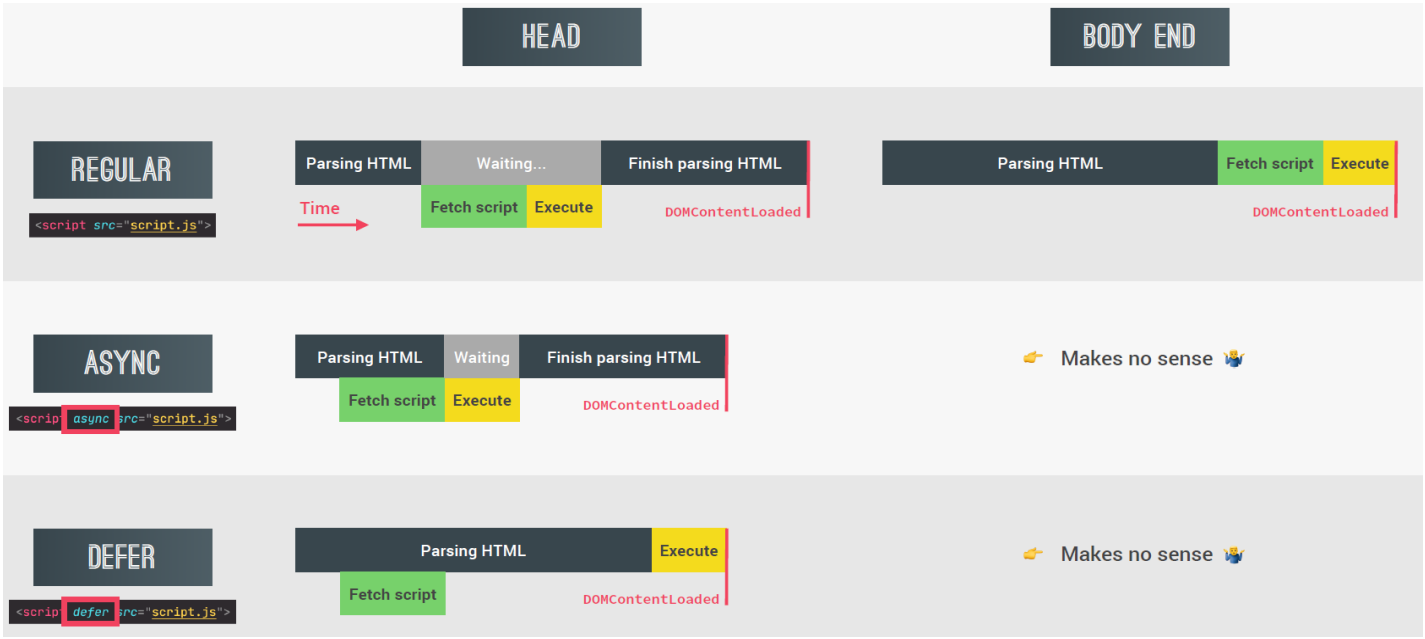
To include the external JavaScript file we will use the script tag in HTML with attribute 'src' as shown in the below-given code-snippet:

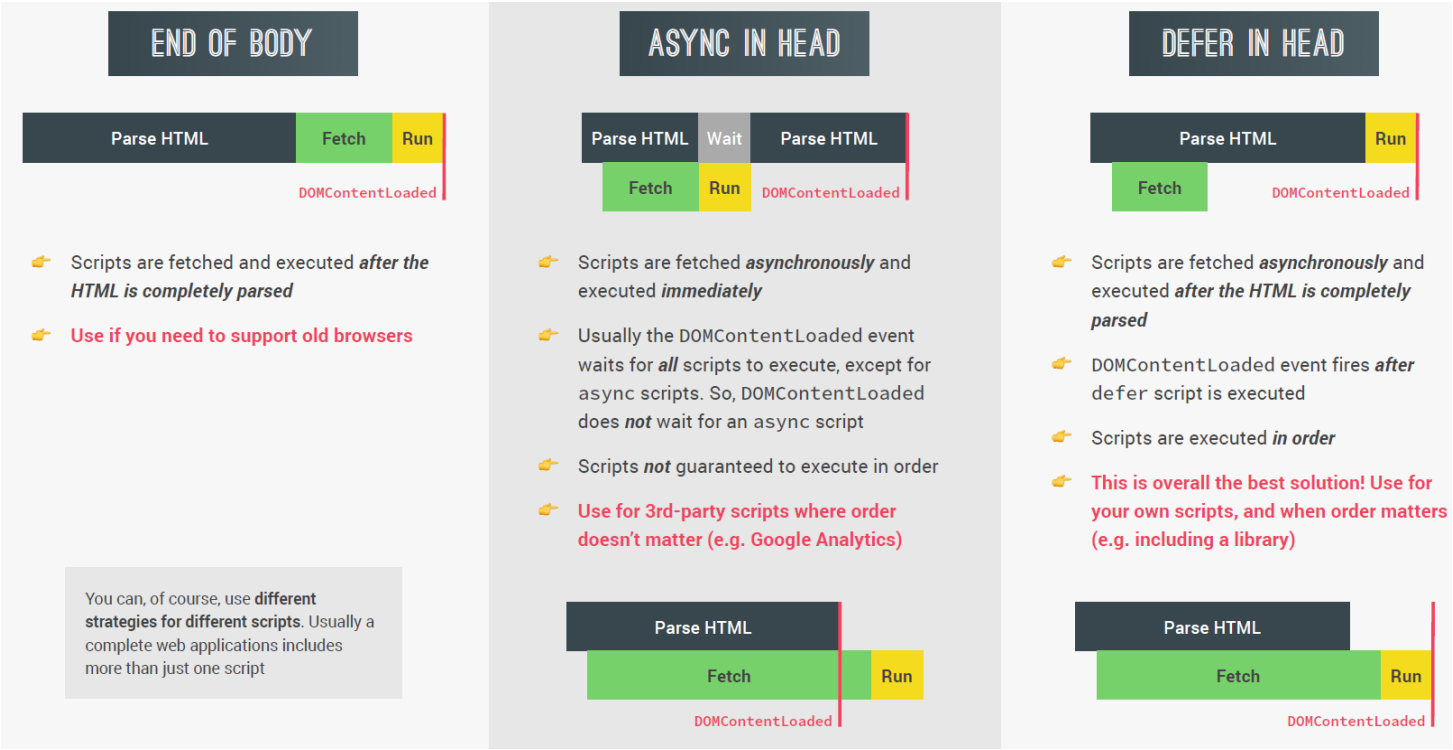
```
<html>
<head>
   <!-- *.js file contain the JavaScript code -->
   <script src="*.js"></script>
</head>
<body>
</body>
</html>
```

1.2.3 Internal Scripting vs External Scripting

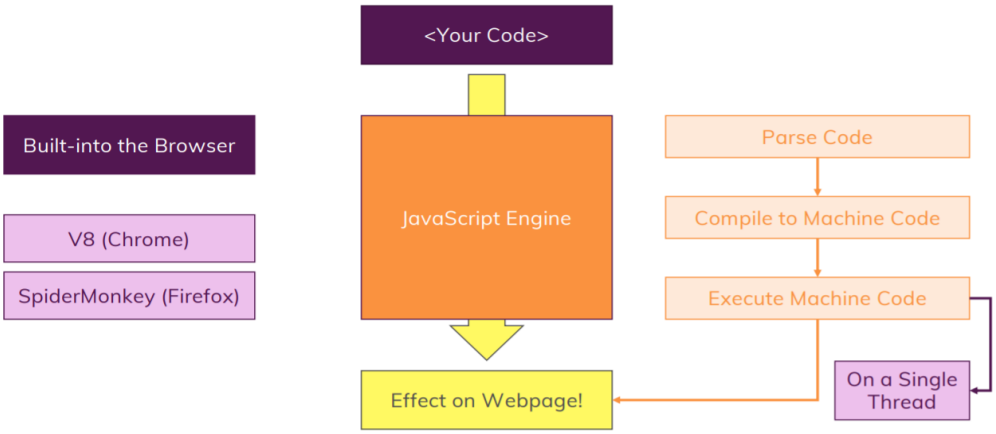
	Internal Scripting	External Scripting
Loading time	Faster, as it is written within the HTML page	Slower, as it is loaded from server, whenever requested
When to use?	If number of lines of code is less	For large amount of code
Re-usable	No, you cannot re-use JavaScript code with any other HTML file	Yes, Same JavaScript file can be used in multiple HTML files
Maintainabilty	Difficult, as for every change request, each HTML pages containing JavaScript code has to be modified separately	Easy, as only 1 file needs to be modified

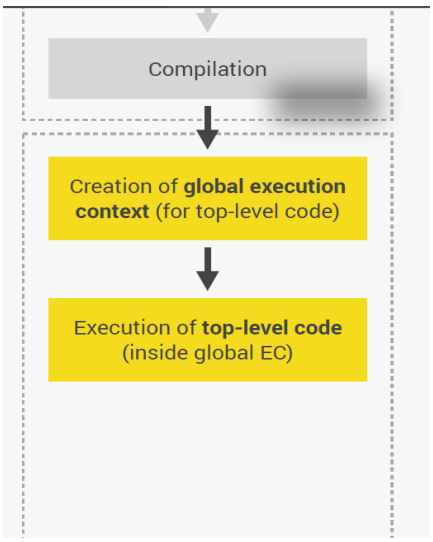
1.2.4 Async, Defer and Regular





1.3 How is JavaScript executed?





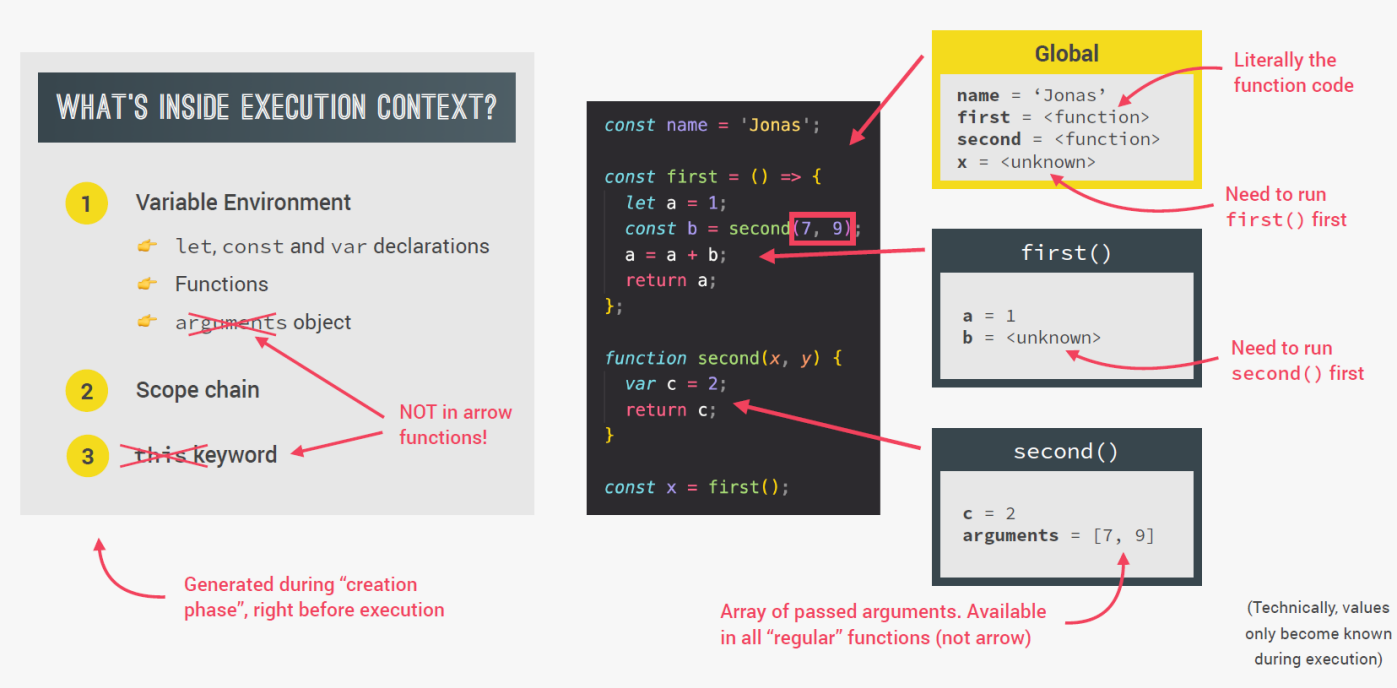
In JavaScript, execution context is an abstract concept that holds information about the environment within which the current code is being executed.

Remember: the JavaScript engine creates the global execution context before it starts to execute any code. From that point on, a new execution context gets created every time a function is executed, as the engine parses through your code. In fact, the global execution context is nothing special. It's just like any other execution context, except that it gets created by default.

When a new execution context is created on a function call, the JavaScript engine needs to spend a little bit of time to configure it, in preparation for the execution. This is essentially what I have been referring to as the memory creation phase in my other articles.

The following happens during this phase.

- Creation of a scope
- Creation of a scope chain
- Determination of the value of this



1.4 Identifiers

Identifiers should follow below rules:

- The first character of an identifier should be letters of the alphabet or an underscore (_) or dollar sign ($).
- Subsequent characters can be letters of alphabets or digits or underscores (_) or a dollar sign ($).
- Identifiers are case-sensitive. Hence, firstName and FirstName are not the same.

Reserved keywords are part of programming language syntax and cannot be used as identifiers.

Types of Identifiers

1.4.1 let

An identifier declared using ‘let’ keyword has a block scope i.e., it is available only within the block in which it is defined.
The value assigned to the identifier can be done either at the time of declaration or later in the code and can also be altered further.

When to use ‘let’ ?

- When you don't need the variable outside the code block.
- When you are looping, and looping variables are never used outside the block.
- To restrict the scope to the block level.
- When the value assigned to the variable can change.

1.4.2 const

The identifier that we choose to hold data that does not vary is called Constant and to declare a constant we use the 'const' keyword followed by an identifier. The value is initialized during the declaration itself and cannot be altered later.

When we talk about the scope of the identifiers declared using the ‘const keyword, it takes the block scope i.e., they exist only in the block of code within which they are defined.

When to use ‘const’ ?

- const is to be used in place of ‘let’ when the variable value should remain constant and shouldn’t be allowed to change throughout the program.
- const has same scope as that of ‘let’ i.e. block scope.

1.4.3 var

The identifiers that we declare to hold data that vary are called Variables and to declare a variable, we optionally use the 'var' keyword.
The value for the same can be initialized optionally. Once the value is initialized, it can be modified any number of times later in the program.
Talking about the scope of the identifier declared using ‘var’ keyword, it takes the Function scope i.e., it is globally available to the Function within which it has been declared and it is possible to declare the identifier name a second time in the same function.

Var	**Let and Const**
Globally available in the function where it is declared	Only available in the code block where it is declared
Hoisted to the top of the function	Not hoisted to the top of the block
Variable name can be declared second time in the same function	Can be declared only once per block

1.5 Hoisting

Hoisting means all the variable and function declarations wherever they are present throughout our program, get lifted and declared to the top of the program. Only the declaration and not the initialization gets hoisted to the top.

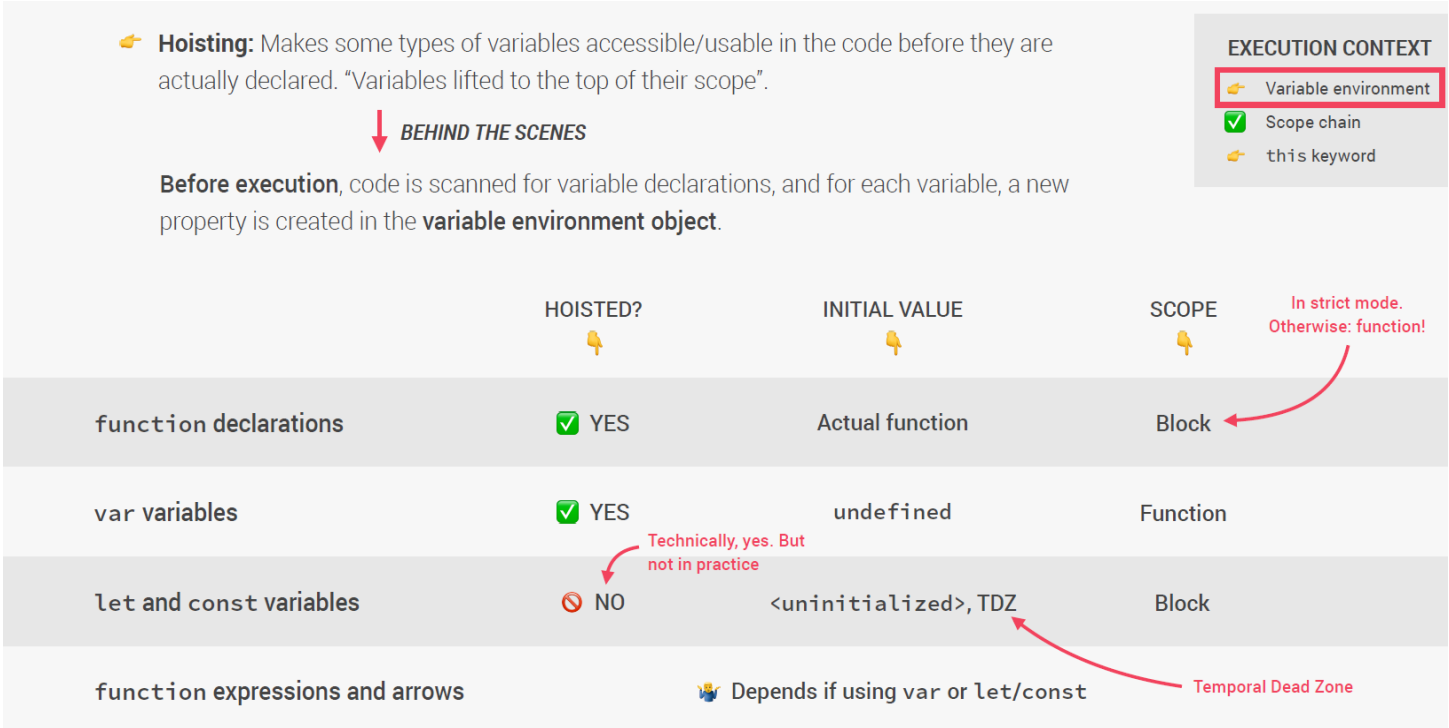
If we try to access a variable without declaration, the Reference Error is thrown.

Variables declared using let and const are not hoisted to the top of the program.

```
console.log("First name: "+firstName); //First name: undefined
var firstName = "Mark";
```

```
console.log("First name: "+firstName);
let firstName = "Mark";
```

The above code throws an error as ”Uncaught ReferenceError: Cannot access 'firstName' before initialization”



1.6 Scope

Accessibility or Visibility of user defined data storage options in the code is called as scope of that particular object.
Its one of the importance feature of JavaScript which avoids naming conflicts and internally system provides automatic memory management.

1.6.1 Global scope

A variable is said to be in global scope when it is accessible

- throughout the program,
- across functions,
- across files

```
//Global variable
var greet = "Hello JavaScript";
function message() {

        //Global variable accessed inside the function
        console.log("Message from inside the function: " + greet);
}
message();
//Global variable accessed outside the function
console.log("Message from outside the function: " + greet);
//Message from inside the function: Hello JavaScript
//Message from outside the function: Hello JavaScript
```

1.6.2 Function scope

Local scope is when a variable is accessible only within a function. This is also called function scope.
In JavaScript, any variable declared with the var keyword inside a function, is considered local. However, if the variable is created with var outside a function, it still behaves like a global variable.

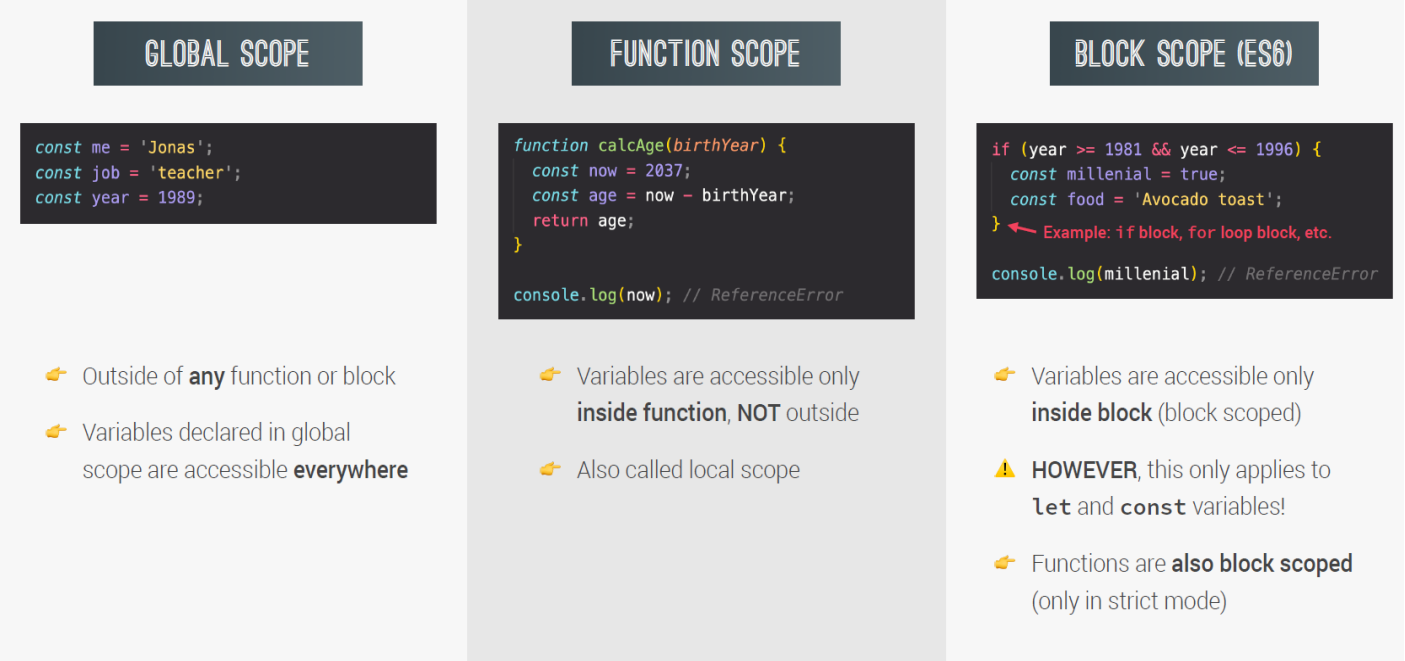
```
function message() {
        //Local variable
        var greet = "Hello JavaScript";
        //Local variables are accessible inside the function
        console.log("Message from inside the function: " + greet);
}
message();
//Local variable cannot be accessed outside the function
console.log("Message from outside the function: " + greet);
//Message from inside the function: Hello JavaScript
//Uncaught ReferenceError: greet is not defined
```

1.6.3 Block scope

A variable with a block scope is accessible only within the block of statements and not throughout the function.

let and const are block-scoped and they exist only in the block in which they are defined.

```
function testVar() {
    if (10 == 10) {
        let flag = "true";
    }
    console.log(flag); //Uncaught ReferenceError: flag is not defined
}
testVar();
```



1.7 Datatypes

Data type mentions the type of value assigned to a variable.
In JavaScript, the type is not defined during variable declaration. Instead, it is determined at run-time based on the value it is initialized with.
Hence, we can say that JavaScript language is a loosely typed or dynamically typed language.

1.7.1 Primitive datatype

The data is said to be primitive if it contains an individual value. Let us explore each of the primitive data types individually.

Number

To store a variable that holds a numeric value, the primitive data type number is used. In almost all the programming languages a number data type gets classified as shown below:

But in JavaScript. the data type number is assigned to the values of type integer, long, float, and double. For example, the data type for values 300, 20.50, 10001, and 13456.89 all are considered to be of the number data type.

In JavaScript, any other value that does not belong to the above-mentioned types is not considered as a legal number. Such values are represented as NaN (Not-a-Number).

String

When a variable is used to store textual value, a primitive data type string is used. Thus, the string represents textual values. String values are written in quotes, either single or double.

Single quotes or double quotes have no special semantics over the other. Though you must be strategic in the selection between the two when you must use one string within the other.

Now, to access any character from within the string we need to be aware of its position in the string. Each character in the string occupies a position.

The first character exists at index 0, next at index 1, and so on.

Boolean

When a variable is used to store a logical value that can be only true or false at all times, primitive data type boolean is used. Thus, boolean is a data type which represents only two values.

Values such as 100, -5, “Cat”, 10<20, 1, 10*20+30, etc. evaluates to true whereas 0, “”, NaN, undefined, null, etc. evaluates to false.

The following values are always falsy:

- false
- 0 (zero)
- -0 (minus zero)
- 0n (BigInt zero)
- '', "", `` (empty string)
- null
- undefined
- NaN

Everything else is truthy. That includes:

- '0' (a string containing a single zero)
- 'false' (a string containing the text “false”)
- [] (an empty array)
- {} (an empty object)
- function(){} (an “empty” function)

Undefined

When the variable is used to store "no value", primitive data type undefined is used.
undefined is a data type in JavaScript that has a single value also termed as undefined.
The undefined value represents "no value".
Any variable that has not been assigned a value during declaration will be automatically assigned with the value undefined.

```
let custName = "John"; //here value is John and the Datatype is String
custName = undefined; //here value and the Datatype are undefined
```

null

The null value represents "no object".

If you are wondering why we would need such a data type, the answer is JavaScript variable intended to be assigned with the object at a later point in the program can be assigned null during the declaration.

```
let item = null;

// variable item is intended to be assigned with object later. Hence null is assigned during variable declaration.
```

BigInt

BigInt is a special numeric type that provides support for integers of random length.

A BigInt is generated by appending n to the end of an integer literal or by calling the function BigInt that generates BigInt from strings, numbers, etc.

```
const bigintvar = 6742347823468988789474747238947782364 7n;
OR
const bigintvar = BigInt("6742347823468988789474747238947782364 7");
const bigintFromNumber = BigInt(10); // same as 10n
```

common math operations can be done on BigInt as regular numbers. But we cannot mix BigInt and regular numbers in the expression.

Type coercion and type conversion

Type coercion is the automatic or implicit conversion of values from one data type to another (such as strings to numbers). Type conversion is similar to type coercion because they both convert values from one data type to another with one key difference — type coercion is implicit whereas type conversion can be either implicit or explicit.

```
const value1 = '5';
const value2 = 9;
let sum = value1 + value2;
console.log(sum);
```

In the above example, JavaScript has coerced the 9 from a number into a string and then concatenated the two values together, resulting in a string of 59. JavaScript had a choice between a string or a number and decided to use a string.

The compiler could have coerced the 5 into a number and returned a sum of 14, but it did not. To return this result, you'd have to explicitly convert the 5 to a number using the Number() method:

```
sum = Number(value1) + value2;
```

1.7.2 Non-Primitive datatype

The data is said to be non-primitive if it contains a reference to the values.

Objects

Objects in JavaScript are a collection of properties and are represented in the form of [key-value pairs].
The 'key' of a property is a string or a symbol and should be a legal identifier.
The 'value' of a property can be any JavaScript value like Number, String, Boolean, or another object.
JavaScript provides the number of built-in objects as a part of the language and user-defined JavaScript objects can be created using object literals.

```
{
  key1 : value1,
  key2 : value2,
  key3 : value3
};
```

Array

The Array is a special data structure that is used to store an ordered collection, which we cannot achieve using the objects.

There are two ways of creating an array:

```
let dummyArr = new Array();
//OR
let dummyArr = [];
```

1.8 Operators

Operators in a programming language are the symbols used to perform operations on the values.

1.8.1 Arthitmetic

```
let sum = 5 + 3; // sum=8
let difference = 5 – 3; // difference=2
let product = 5 * 3; // product=15
let division = 5/3; // division=1
let mod = 5%3; // mod=2
let expo=5**3;//expo=125
let value = 5;
value++; // increment by 1, value=6
let value = 10;
value--; // decrement by 1, value=9
```

Arithmetic operator '+' when used with string type results in the concatenation.

```
let firstName = "James";
let lastName = "Roche";
let name = firstName + " " + lastName;  // name = James Roche
```

Arithmetic operator '+' when used with a string value and a numeric value, it results in a new string value.

```
let strValue="James";
let numValue=10;
let newStrValue= strValue + " " + numValue;  // newStrValue= James 10
```

1.8.2 Assignment

```
let num = 30; // num=30
let num += 10; // num=num+10 => num=40
let num -= 10; // num=num-10 => num=20
let num *= 30; // num=num*30 => num=900
let num /= 10; // num=num/10 => num=3
let num %= 10; // num=num%10 => num=0
```

1.8.3 Relational

```
10 > 10; //false
10 >= 10; //true
10 < 10; //false
10 <= 10; //true
10 == 10; //true
10 != 10; //false
```

Double Equals (==)

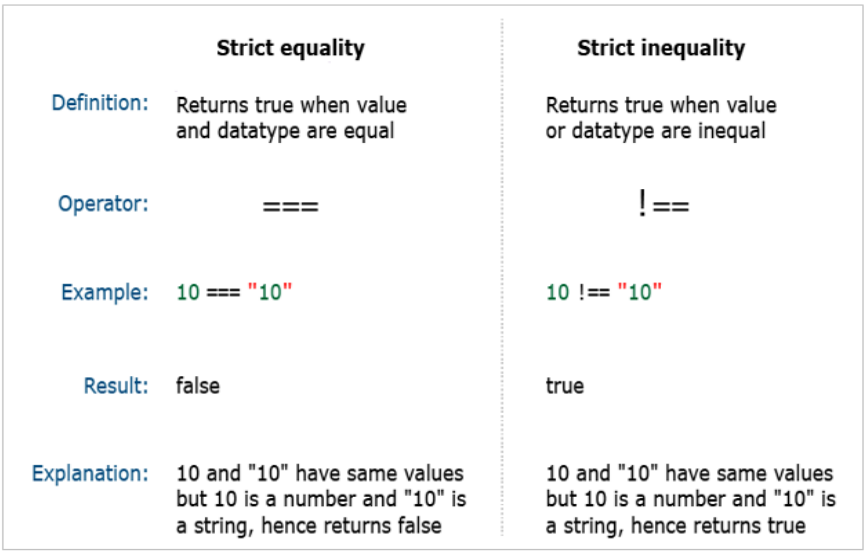
Double equals compares the values only, irrespective of the data types. For example:

```
console.log('100' == 100)   // string & number
// true
var x = 5;   // number
var y = '5' // string
console.log(x == y)    // true
```

Triple Equals (===)

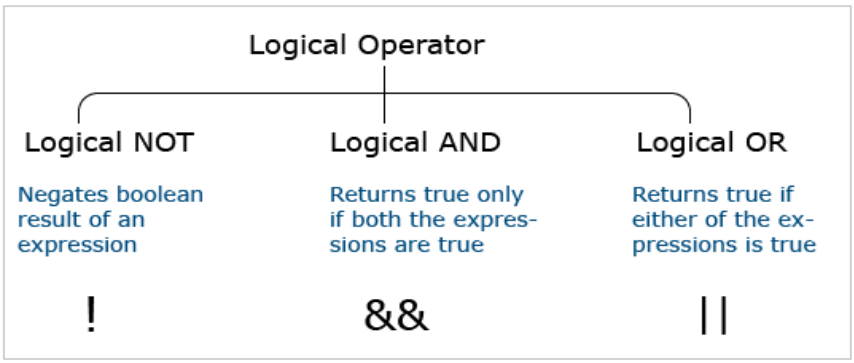
Triple equals test strict equality which means value and type both should be same. For example:

```
console.log('100' === 100)   // string & number
// false
var x = 5; // number
var y = '5' // string
console.log(x === y)  // false
```



	Strict equality	Strict inequality
Definition:	Returns true when value and datatype are equal	Returns true when value or datatype are inequal
Operator:	===	!==
Example:	10 === "10"	10 !== "10"
Result:	false	true
Explanation:	10 and "10" have same values but 10 is a number and "10" is a string, hence returns false	10 and "10" have same values but 10 is a number and "10" is a string, hence returns true

1.8.4 Logical



```
!(10 > 20); //true
(10 > 5) && (20 > 20); //false
(10 > 5) || (20 > 20); //true
```

1.8.5 Unary

"typeof" is an operator in JavaScript.

JavaScript is a loosely typed language i.e., the type of variable is decided at runtime based on the data assigned to it. This is also called dynamic data binding.

As programmers, if required we can use this operator typeof to find the data type of a JavaScript variable.

The following are the ways in which it can be used and the corresponding results that it returns.

```
typeof "JavaScript World" //string
typeof 10.5 // number
typeof 10 > 20 //boolean
typeof undefined //undefined
typeof null //Object
typeof {itemPrice : 500} //Object
```

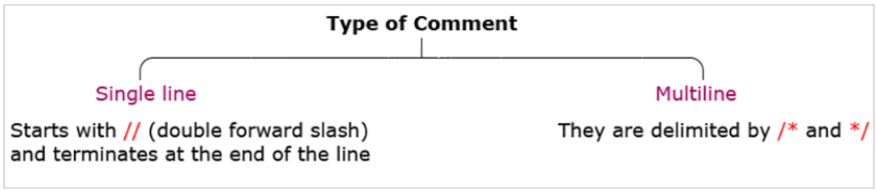
1.9 Statements

Statements are instructions in JavaScript that have to be executed by a web browser. JavaScript code is made up of a sequence of statements and is executed in the same order as they are written.

1.9.1 Non-Conditional Statements

Comment

At times, we may not want to execute a certain portion of our code or maybe we want to add information in our code that explains the significance of the line of code being written.
Well, when we want to exclude some part of our code from getting executed, comment statements help us.



Break

While we are iterating over the block of code getting executed within the loop, we may want to exit the loop if a certain condition is met.

'break' statement is used to terminate the loop and transfer control to the first statement following the loop.

```
var counter = 0;
for (var loop = 0; loop < 5; loop++) {
   if (loop == 3)
      break;
   counter++;
}
```

loopVar	counter
0	1
1	2
2	3
3	Loop terminated. counter = 3.

The if statement used in the above example is a conditional / decision-making statement.

Continue

There are times when during the iteration of the block of code within the loop, we may want to skip the block execution for a specific value and then continue executing the block for all the other values. JavaScript gives us a 'continue' statement to handle this.

Continue statement is used to terminate the current iteration of the loop and continue execution of the loop with the next iteration.

```
var counter = 0;
for (var loop = 0; loop < 5; loop++) {
   if (loop == 3)
      continue;
   counter++;
}
```

loopVar	Counter
0	1
1	2
2	3
3	Iteration terminated. Hence counter is not incremented.
4	4

The if statement used in the example is a conditional / decision-making statement.

1.9.2 Conditional Statements

If -else

The 'if' statement is used to execute a block of code if the given condition evaluates to true.

else statement is used to execute a block of code if the given condition evaluates to false.

```
let num1 = 1;
if (num1 % 2 == 0) {
   console.log("It is an even number!!");
}
else{
   console.log("It is an odd number!!");
}
//OUTPUT: It is an odd number!! Because in if 1%2 evaluates to false and moves to else condition
```

if...else ladder is used to check for a new condition when the first condition evaluates to false.

```
if (condition1) {
   // block of code that will be executed if condition1 is true
}
else if (condition2) {
   // block of code that will be executed if the condition1 is false and condition2 is true
}
else {
   // block of code that will be executed if the condition1 is false and condition2 is false
}
```

Switch

The Switch statement is used to select and evaluate one of the many blocks of code.

```
switch (expression) {
   case value1: code block;
            break;
   case value2: code block;
            break;
   case valueN: code block;
            break;
   default: code block;
}
```

'break' statement is used to come out of the switch and continue execution of statement(s) the following switch.

Looping statements in JavaScript helps to execute statement(s) required number of times without repeating code.

Ternary operator

It is a conditional operator that evaluates to one of the values based on whether the condition is true or false.

It happens to be the only operator in JavaScript that takes three operands. It is mostly used as a shortcut of 'if-else' condition.

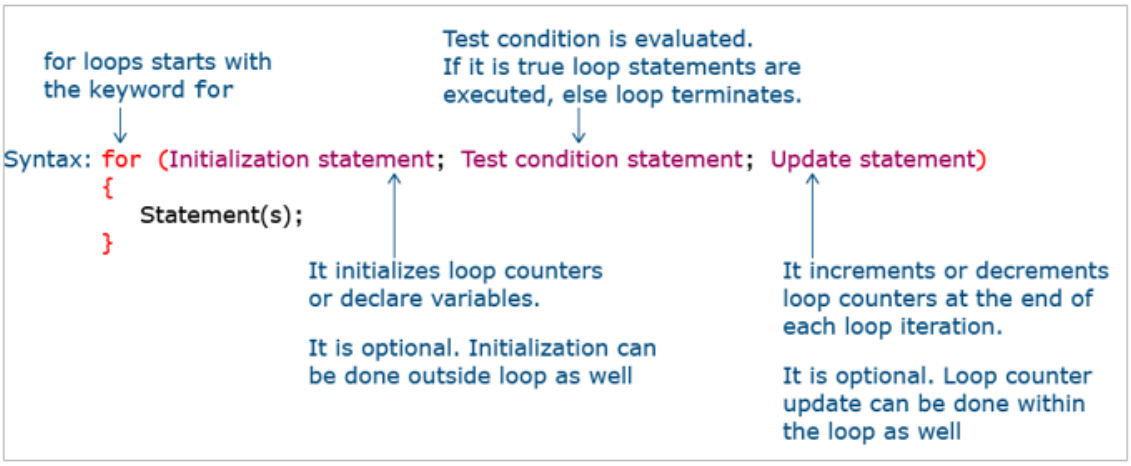
```
let workingHours = 9.20;
let additionalHours;
(workingHours > 9.15) ? additionalHours = "You have positive additional hours" : additionalHours =
"You have negative additional hours";
console.log(additionalHours);
```

1.10 Loops

Looping statements in JavaScript helps to execute statement(s) required number of times without repeating code.

for

'for' loop is used when the block of code is expected to execute for a specific number of times. To implement it, use the following syntax.

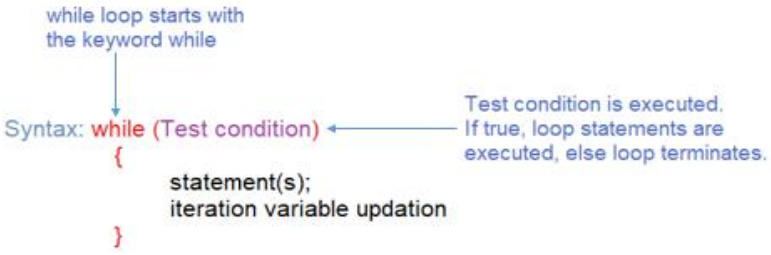



```
let counter = 0;
for (let loopVar = 0; loopVar < 5; loopVar++) {
        counter = counter + 1;
        console.log(counter);
}
```

loopVar	counter
0	1
1	2
2	3
3	4
4	5

while

'while' loop is used when the block of code is to be executed as long as the specified condition is true. To implement the same, we use the following syntax:

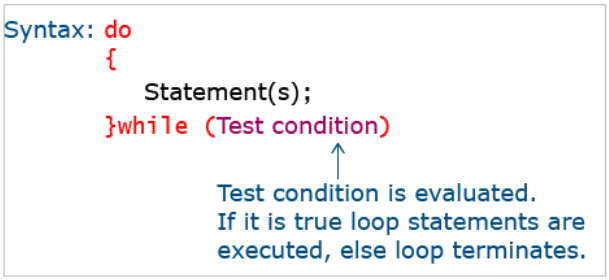



```
let counter = 0;
let loopVar = 0;
while (loopVar < 5) {
        console.log(loopVar);
        counter++;
        loopVar++;
        console.log(counter);
}
```

loopVar	counter
0	1
1	2
2	3
3	4
4	5

do- while

'do-while' is a variant of 'while' loop.
This will execute a block of code once before checking any condition.
Then, after executing the block it will evaluate the condition given at the end of the block of code.
Now the statements inside the block of code will be repeated till condition evaluates to true.



```
let counter = 0;
let loopVar = 0;
do {
        console.log(loopVar);
        counter++;
        loopVar++;
        console.log(counter);
}
while (loopVar < 5);
```

1.11 Template Literals

A template literal is a new type of string literal in ES6.

It can span multiple lines and interpolate expressions to include their results.

```
let firstName = 'Marty';
let lastName = 'Hall';
console.log(`Name: ${firstName} ${lastName}
Email: ${firstName}_${lastName}@abc.com`);

//Will display the below output
Name: Marty Hall
Email: Marty_Hall@abc.com
```

Same code in ES5 will comprise of the string in “” and concatenating static content with variable expressions with + and \n for newline

The template literal notation of ES6 enclosed in `` makes it convenient to have multiline statements with expressions.

The variables are accessed using ${} notation.

2. Array

Array in JavaScript is an object that allows storing multiple values in a single variable. An array can store values of any datatype. An array's length can change at any time, and data can be stored at non-contiguous locations in the array.

2.1 Creation of Arrays

2.1.1 Array Literal Notation

```
let myArray = [element 1, element2,…, element N];
```

2.1.2 Array Constructor

Arrays can be created using the Array constructor with a single parameter which denotes the array length. The parameter should be an integer between 0 and 232-1 (inclusive). This creates empty slots for the array elements. If the argument is any other number, a RangeError exception is thrown.

```
let myArray = new Array(arrayLength);
```

If more than one argument is passed to the Array constructor, a new Array with the given elements is created.

```
let myArray = new Array(element 1, element 2,…,element N);
```

2.1.3 Array.from()

Array.from() lets you create Arrays from:

- array-like objects (objects with a length property and indexed elements); or
- iterable objects (objects such as Map and Set).
-

Array.from() has an optional parameter mapFn, which allows you to execute a map() function on each element of the array being created.
More clearly, Array.from(obj, mapFn, thisArg) has the same result as Array.from(obj).map(mapFn, thisArg), except that it does not create an intermediate array, and *mapFn* only receives two arguments (*element*, *index*).

```
Array.from('foo'); // [ "f", "o", "o" ]

const set = new Set(['foo', 'bar', 'baz', 'foo']);
Array.from(set); // [ "foo", "bar", "baz" ]


const map = new Map([[1, 2], [2, 4], [4, 8]]);
Array.from(map);
// [[1, 2], [2, 4], [4, 8]]
const mapper = new Map([['1', 'a'], ['2', 'b']]);
```

```
Array.from(mapper.values());
// ['a', 'b'];

Array.from(mapper.keys());
// ['1', '2'];

// Using an arrow function as the map function to
// manipulate the elements
Array.from([1, 2, 3], x => x + x);
// [2, 4, 6]

// Generate a sequence of numbers
// Since the array is initialized with `undefined` on each position,
// the value of `v` below will be `undefined`
Array.from({length: 5}, (v, i) => i);
// [0, 1, 2, 3, 4]
```

2.2 Destructuring Arrays

JavaScript introduced the destructuring assignment syntax that makes it possible to unpack values from arrays or objects into distinct variables. Let's see how this syntax helps us to unpack values from an array.

```
// [RN1] we have an array with the employee name and id
let empArr = ["Shaan", 104567];
// destructuring assignment
// sets empName = empArr[0]
// and empId = empArr[1]
let [empName, empId] = empArr;
console.log(empName); // Shaan
console.log(empId);  // 104567
```

```
let [empName, , location] = ["Shaan", 104567, "Bangalore"];

//Here second element of array is skipped and third element is assigned to location variable

console.log(empName); // Shaan

console.log(location);  // Bangalore
```

Rest operator can also be used with destructuring assignment syntax.

```
let [empName, ...rest] = ["Shaan", 104567, "Bangalore"];
console.log(empName); // Shaan
console.log(rest);  // [104567,'Bangalore']
```

Here, the value of the rest variable is the array of remaining elements and the rest parameter always goes last in the destructuring assignment.

2.3 Combining and Cloning Arrays using Spread operator

Spread operator is a new operator that was introduced as part of JavaScript in 2015. It consists of triple dots (...) which helps in spreading out the elements of an array to a new variable.
When the spread operator is used in the function call, it expands the iterable object, i.e., array into the list of arguments.

```
let numArr = [10, 5, 20];
//spread turns array into the list of arguments
console.log(Math.max(...numArr)); // 20
```

We can merge arrays using the spread syntax.

```
let arr1 = [3, 5, 1];
let arr2 = [8, 2, 6];
let newArr = [...arr1, ...arr2];
console.log(newArr); // [3,5,1,8,2,6]
```

We can even combine arrays with normal values.

```
let arr1 = [3, 5, 1];
let arr2 = [8, 2, 6];
let newArr = [0, ...arr1, 4, ...arr2];
console.log(newArr); // [0,3,5,1,4,8,2,6]
```

You can also use the spread operator to create a copy of an array.

```
let arr1 = [3, 5, 1];
let arrCopy = [...arr1];
arrCopy.push(4);
console.log(arrCopy);
//arrCopy becomes [3,5,1,4] and arr1 remains unaffected
```

2.4 Accessing Arrays

Array elements can be accessed using indexes. The first element of an array is at index 0 and the last element is at the index equal to the number of array elements – 1. Using an invalid index value returns undefined.

```
let arr = ["first", "second", "third"];
console.log(arr[0]); //first
console.log(arr[1]); //second
console.log(arr[3]); //undefined
```

Loop over an array

You can loop over the array elements using indexes.

```
let colors = ["Red", "Orange", "Green"];
for (let i = 0; i < colors.length; i++) {
   console.log(colors[i]);
}
//Red
//Orange
//Green
```

JavaScript also provides for..of statement to iterate over an array.

```
let colors = ["Red", "Orange", "Green"];
// iterates over array elements
for (let color of colors) {
   console.log(color);
}
//Red
//Orange
//Green
```

- It avoids all the pitfalls of for–in and can be used with any collection.
- Unlike forEach(), it works with break, continue, and return.
- for ...of is not just for arrays. It also works on most array-like objects, like DOM NodeLists.
- It also works on Map and Set objects introduced newly in ES6.
- for …of works with several different data structures provided by the language and its standard library.
- for …of can be used to iterate over any iterable object.

2.5 Array Methods

Methods to add/remove Array elements

2.5.1 push()

Adds new element to the end of an array and return the new length of the array.

```
let myArray = ["Android", "iOS", "Windows"];
myArray.push("Linux");
console.log(myArray);
// ["Android","iOS","Windows","Linux"]
```

2.5.2 pop()

Removes the last element of an array and returns that element.

```
let myArray = ["Android", "iOS", "Windows"];
console.log(myArray.pop()); // Windows
console.log(myArray); // ["Android","iOS"]
```

2.5.3 shift()

Removes the first element of an array and returns that element.

```
let myArray = ["Android", "iOS", "Windows"];
console.log(myArray.shift()); //Android
console.log(myArray); //["iOS", "Windows"]
```

2.5.4 unshift()

Adds new element to the beginning of an array and returns the new length.

```
let myArray = ["Android", "iOS", "Windows"];
myArray.unshift("Linux");
console.log(myArray);//["Linux","Android","iOS","Windows"]
```

2.5.5 splice()

Change the content of an array by inserting, removing, and replacing elements. Returns the array of removed elements.

Syntax:

```
array.splice(index,deleteCount,items);
index = index for new item
deleteCount = number of items to be removed, starting from index next to index of new item
items = items to be added
```

```
let myArray = ["Android", "iOS", "Windows"];
//inserts at index 1
myArray.splice(1, 0, "Linux");
console.log(myArray);
// ["Android","Linux", "iOS", "Windows"]
```

2.5.6 slice()

Returns a new array object copying to it all items from start to end(exclusive) where start and end represents the index of items in an array. The original array remains unaffected.

Syntax:

```
array.slice(start,end)
```

```
let myArray=["Android","iOS","Windows"];
console.log(myArray.slice(1,3)); // ["iOS", "Windows"]
```

2.5.7 concat()

Joins two or more arrays and returns joined array.

```
let myArray1 = ["Android","iOS"];
let myArray2 =  ["Samsung", "Apple"];
console.log(myArray1.concat(myArray2));
//["Android", "iOS", "Samsung", "Apple"]
```

Methods to search among Array elements

2.5.8 find()

Returns the value of the first element in an array that passes a condition specified in the callback function.

Else, returns undefined if no element passed the test condition.

Syntax:

```
array.find(callback(item,index,array))
callback is a function to execute on each element of the array
item value represents the current element in the array
index value indicates index of the current element of the array
array value represents array on which find() is used,
index and array are optional
```

```
let myArray = ["Android", "iOS", "Windows", "Linux"];
let result = myArray.find(element => element.length > 5);
console.log(result); //Android
```

2.5.9 indexOf()

Returns the index for the first occurrence of an element in an array and -1 if it is not present.

```
let myArray = ["Android","iOS","Windows","Linux"];
console.log(myArray.indexOf("iOS")); // 1
console.log(myArray.indexOf("Samsung")); //-1
```

2.5.10 findIndex()

Returns the index of the first element in an array that passes a condition specified in the callback function. Returns -1 if no element passes the condition.

Syntax:

```
Array.findIndex(callback(item,index,array));
callback is a function to execute on each element of the array
item value represents current element in the array
index represents index of the current element of the array
array represents array on which findIndex() is used.
index and array are optional
```

```
let myArray = ["Android", "iOS", "Windows", "Linux"];
let result = myArray.findIndex(element => element.length > 5);
console.log(result) //0
```

2.5.11 filter()

Creates a new array with elements that passes the test provided as a function.

Syntax:

```
array.filter(callback(item,index,array))
callback is the Function to test each element of an array
item value represents the current element of the array
index value represents Index of current element of the array
array value indicates array on which filter() is used.
```

```
let myArray = ["Android", "iOS", "Windows", "Linux"];
let result = myArray.filter(element => element.length > 5);
console.log(result)
//["Android","Windows"]
```

Method to iterate over Array elements.

2.5.12 forEach()

Iterates over an array to access each indexed element inside an array.

Syntax:

```
array,forEach(callback(item,index,array))
callback is a function to be executed on each element of an array
item value represents current element of an array
index value mentions index of current element of the array
array represents the array on which forEach() is called
```

```
let myArray = ["Android", "iOS", "Windows"];
myArray.forEach((element, index) =>
console.log(index + "-" + element));
//0-Android
//1-iOS
//2-Windows
//3-Linux
```

Continue and Break statements have no effect.

Methods to transform the Array

2.5.13 map()

Creates a new array from the results of the calling function for every element in the array.

Syntax:

```
array.map(callback(item,index,array))
callback is a function to be run for each element in the array
item represents the current element of the array
index value represents index of the current element of the array
array value represents array on which forEach() is invoked
```

```
let numArr = [2, 4, 6, 8];
let result = numArr.map(num=>num/2);
console.log(result);//[ 1, 2, 3, 4 ]
```

2.5.14 reduce()

Executes a defined function for each element of passed array and returns a single value

Syntax:

```
array.reduce(callback(accumulator, currentValue, index,array),initialValue)
callback is a function to be executed on every element of the array
accumulator is the initialValue or previously returned value from the function.
currentValue represents the current element of the passed array
index represents index value of the current element of the passed array
array represents the array on which this method can be invoked.
initialValue represents the Value that can be passed to the function as an initial value.
currentValue,index,array and initialValue are optional.
```

```
const numArr = [1, 2, 3, 4];
// 1 + 2 + 3 + 4
console.log(numArr.reduce(
(accumulator, currentVal) =>
accumulator + currentVal));
// 10
// 5 + 1 + 2 + 3 + 4
console.log(numArr.reduce(
(accumulator, currentVal) =>
accumulator + currentVal,5));
// 15
```

2.5.15 join()

Returns a new string by concatenating all the elements of the array, separated by a specified operator such as comma. Default separator is comma.

```
let myArray = ["Android", "iOS", "Windows"];
console.log(myArray.join());
// Android,iOS,Windows
console.log(myArray.join('-'));
 // Android-iOS-Windows
```

2.5.16 sort()

The sort() method sorts the elements of an array in place and returns the sorted array. The default sort order is ascending, built upon converting the elements into strings, then comparing their sequences of UTF-16 code units values.

```
// Functionless
sort()
// Arrow function
sort((firstEl, secondEl) => { ... } )
// Compare function
sort(compareFn)
// Inline compare function
sort(function compareFn(firstEl, secondEl) { ... })
```

```
const months = ['March', 'Jan', 'Feb', 'Dec'];
months.sort();
console.log(months);
// expected output: Array ["Dec", "Feb", "Jan", "March"]

const array1 = [1, 30, 4, 21, 100000];
array1.sort();
console.log(array1);
// expected output: Array [1, 100000, 21, 30, 4]
```

If compareFunction is supplied, all non-undefined array elements are sorted according to the return value of the compare function (all undefined elements are sorted to the end of the array, with no call to compareFunction). If a and b are two elements being compared, then:

- If compareFunction(a, b) returns a value > than 0, sort b before a.
- If compareFunction(a, b) returns a value < than 0, sort a before b.
- If compareFunction(a, b) returns 0, a and b are considered equal.

Sorting non-ASCII characters

For sorting strings with non-ASCII characters, i.e. strings with accented characters (e, é, è, a, ä, etc.), strings from languages other than English, use String.localeCompare. This function can compare those characters so they appear in the right order.

```
var items = ['réservé', 'premier', 'communiqué', 'café', 'adieu', 'éclair'];
items.sort(function (a, b) {
  return a.localeCompare(b);
});

// items is ['adieu', 'café', 'communiqué', 'éclair', 'premier', 'réservé']
```

2.5.17 flat()

The flat() method creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.

```
const arr1 = [1, 2, [3, 4]];
arr1.flat();// [1, 2, 3, 4]

const arr2 = [1, 2, [3, 4, [5, 6]]];
arr2.flat();// [1, 2, 3, 4, [5, 6]]

const arr3 = [1, 2, [3, 4, [5, 6]]];
arr3.flat(2);// [1, 2, 3, 4, 5, 6]

const arr4 = [1, 2, [3, 4, [5, 6, [7, 8, [9, 10]]]]];
arr4.flat(Infinity);// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

2.5.18 reverse()

The reverse() method reverses an array in place. The first array element becomes the last, and the last array element becomes the first.

```
const a = [1, 2, 3];
console.log(a); // [1, 2, 3]
a.reverse();
console.log(a); // [3, 2, 1]
```

3. Set

Set is a new built-in object introduced in ES6 which is similar to an array but ensures distinct values.
Sets are not indexed based and can not be referred based on the position – items in a set can not be accessed individually.
Add/remove and looping over set is permissible.
Both primitives and objects are allowed in a set.

Create an empty set:

```
var courses = new Set();
console.log(courses);
```

Create a set with list of values:

```
let courses = new Set(["Angular","React","Vue","Angular"]);
console.log(courses);
▶ Set(3) {"Angular", "React", "Vue"}
```

Note that the set has automatically removed the duplicate entry while building.

Set Manipulation

We can use the methods add() and delete() to add more elements and remove an existing element from a set.

```
courses.add("Express")
▶ Set(4) {"Angular", "React", "Vue", "Express"}
courses.add("Node")
▶ Set(5) {"Angular", "React", "Vue", "Express", "Node"}
courses.delete("Vue");
true
console.log(courses);
▶ Set(4) {"Angular", "React", "Express", "Node"}
```

Working with Sets

Below properties/methods help in working with sets:

.size

Helps in getting the length of the set.

.has()

Helps in checking if an element exist in a set.

.values()

Fetches all the values in a set.

.clear()

Clears all elements in the set.

Iterating a set

Since set is a built-in object it has a default iterator and hence can be iterated through any of the following methods:

- Using Set's default iterator to step through each item in a Set.
- Using the new for...of loop

The .values() method on a set returns a new iterator object called SetIterator which can be stored in a variable and loop through each item using .next().

```
let myiterator=courses.values()
myiterator.next()
▶ {value: "Angular", done: false}
myiterator.next()
▶ {value: "React", done: false}
myiterator.next()
▶ {value: "Express", done: false}
myiterator.next()
▶ {value: "Node", done: false}
myiterator.next()
▶ {value: undefined, done: true}
```

The for… of loop in ES6 can be used to iterator over any iterable and all the built-ins in ES6 are iterable by default.

```
for(course of courses)
    console.log(course);
Angular
React
Express
Node
```

4. WeakSet

A WeakSet is just like a normal Set with a few key differences:

- a WeakSet can only contain objects.
- a WeakSet is not iterable which means it can't be looped over.
- a WeakSet does not have a .clear() method.
- A WeakSet is created like a normal Set, except that it uses the WeakSet constructor.

```
const employee1 = { name: 'Aadya', age: 26, role: 'UI Developer' };
const employee2 = { name: 'Vidya', age: 27, role: 'Full Stack Developer' };
const employee3 = { name: 'Navya', age: 31, role: 'Web Developer' };

const employees = new WeakSet([employee1, employee2, employee3]);
console.log(employees);
```

Why WeakSet?

- WeakSets take advantage of JavaScript's garbage collection by exclusively working with objects.
- When an object is set to null, then the object is eligible for garbage collection.
- When the JavaScript's garbage collector runs, the memory that object previously occupied will be freed up to be used later in the program.

5. Maps

A Map is similar to an object where data is stored in key-value pairs
Both the keys and the values can be objects, primitive values, or a combination of the two.

```
const employees = new Map();

employees.set('veena_bhat@i.com', {
    firstName: 'Veena',
    lastName: 'Bhat',
    role: 'UI Developer'
});
employees.set('aadya_kamath@i.com', {
    firstName: 'Aadya',
    lastName: 'Kamath',
    role: 'UX Designer'
});
```

Working with Maps

delete()

method can be used to remove a key-value pair

```
employees.delete("veena_bhat@i.com");
true
console.log(employees);
▶ Map(1) {"aadya_kamath@i.com" => {…}}
```

clear()

method will help in removing all the key-value pairs from the map.

has()

allows to check if a key-value pair is available in the map.

get()

helps in fetching the value for a given key.

```
employees.has("aadya_kamath@i.com");
true
employees.get("aadya_kamath@i.com")
▶ {firstName: "Aadya", lastName: "Kamath", role: "UX Designer"}
```

keys()

Returns an array of keys.

values()

Returns an array of values.

.entries()

The entries() method returns a new Iterator object that contains the [key, value] pairs for each element in the Map object in insertion order. In this particular case, this iterator object is also an iterable, so the for-of loop can be used.

Iterating over Maps

There are 3 different ways in which a map can be iterated over:

Using the for… of loop

```
for(const employee of employees)
    console.log(employee);
```

```
▶ (2) ["veena_bhat@i.com", {…}]
▶ (2) ["aadya_kamath@i.com", {…}]
```

For of loop helps in getting the data for each key-value pair and displays in array format where first value of the array is the key of a Map and subsequent will be the value.

```
const courses = new Map();

courses.set('Angular', 1001);
courses.set('React', 1002);
courses.set('Node', 1003);
courses.set('Express', 1004);

for (const course of courses) {
    var [key,value]=course;
    console.log(key, value);
}
```

Using the default iterator of Map – MapIterator

- Maps allow iterating over both keys and values.
- Every time, .keys() and .values() methods on a Map is called, it return a new iterator object called MapIterator.
- The iterator object can be stored in a new variable and use .next() to loop through each key/value.

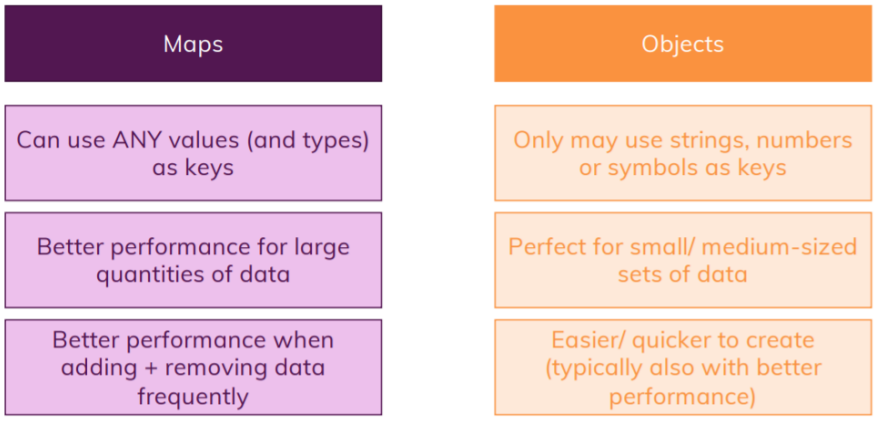
```
let iteratorEmpKeys = employees.keys();
iteratorEmpKeys.next();
```

```
let iteratorEmpKeys = employees.keys();
iteratorEmpKeys.next();
```

Using forEach loop

```
courses.forEach((value,key)=>console.log(key,value));
```

```
Angular 1001
React 1002
Node 1003
Express 1004
```



6. WeakMap

A WeakMap is like a normal Map with a few key differences:

- In a WeakMap only objects are allowed to be used as keys.
- A WeakMap is not iterable.
- A WeakMap does not have a .clear() method.
- A WeakMap can be created similar to a normal Map, except that it uses the WeakMap constructor.

```
const book1 = { title: 'Learning ES6', author: 'Tara MS' };
const book2 = { title: 'Node with Express', author: 'Prajila V.K' };
const book3 = { title: 'Angular with PrimeNG', author: 'Vidya Bhat' };

const library = new WeakMap();
library.set(book1, true);
library.set(book2, false);
library.set(book3, true);
```

For better and easier use and maintainability, WeakMaps leverage garbage collection.

```
book1 = null;
console.log(library);
```

Now the reference book1 is eligible for garbage collection and will be removed from the Map automatically when the JavaScript garbage collector runs making the code more efficient in terms of memory utilization.

7. String

Methods

7.1 charAt()

It retrieves a character that resides on the index passed as an argument.

```
let myString = "Hello World";
console.log("Character at position 4 is : " + myString.charAt(3));
//Returns: Character in position 4 is: l
```

7.2 concat()

It accepts an unlimited number of string arguments, joins them, and returns the combined result as a new string.

```
let myStr1 = "Hello";
let myStr2 = " ";
let myStr3 = "World";
console.log("Concatenated string: "+myStr1.concat(myStr2,myStr3));
//Returns: Concatenated string: Hello World
```

7.3 indexOf()

It returns the index of the given character or maybe the given set of characters in a string passed as an argument.

```
let myString = "Hello World";
console.log("Index of character l is : "+myString.indexOf('l'));
//Returns: Index of character l is : 2
```

7.4 match()

It makes use of the regular expression to look for a specific string and returns all the strings that match.

```
let myStr = "Are you enjoying JavaScript?";
console.log(myStr.match(/you/));
/*Returns an array:
[
  'you',
  index: 4,
  input: 'Are you enjoying JavaScript?',
  groups: undefined
]
```

```
*/
```

7.5 replace()

It accepts the substring or the regular expression as well as the string that will be used for the replacement string. The idea is to replace all matches with the replacement string and provide the modified string.

```
let myStr = "Are you enjoying JavaScript?";
myStr = myStr.replace('you', 'they');
console.log(myStr);
//Returns Are they enjoying JavaScript?
```

7.6 search()

It searches for a match of regular expression in the given string and returns its position. If there is no match, it returns -1.

```
let myString1 = "can you find it?";
console.log("Occurrence of find in statement1: "+myString1.search('find'));
let myString2 = "Or you can not?";
console.log("Occurrence of find in statement2: "+myString2.search('find'));
/*Returns:
 The Occurrence of find in statement1: 8,
 The Occurrence of find in statement2: -1*/
```

7.7 split()

It splits the given string into the array of substrings where separator marks the index for split begin and end. Say, if the string consists of a comma (,) then the given string in the argument will be split at every comma.

```
let myString = "Hello World";
console.log("Split string based on spaces: "+myString.split(" "));
//Returns: Split of string based on spaces: Hello,World
```

7.8 slice()

It extracts and returns part of a string. The Second parameter is optional.
If only one parameter is passed, it is the index from which string will start slicing from till the end of this string. If two parameters are passed, the string between these 2 index values is sliced.
Index value passed as the first parameter is included whereas index value passed as the second parameter is excluded.

```
let myString = "Hello World";
console.log("Slicing using 2 parameters: "+myString.slice(0,5));
console.log("Slicing using 1 parameter: "+myString.slice(5));
/*Returns:
Slicing using 2 parameters: Hello,
Slicing using 1 parameter: World*/
```

7.9 substring()

It extracts and returns part of a string. Compared to the slice() method, it can accept a negative parameter, meaning slicing should start from the end.

```
let myString = "Hello World";
console.log("Substring using 2 parameters: "+myString.substring(2,5));
console.log("Substring using 1 parameter: "+myString.substring(5));
/*Returns:
Substring using 2 parameters: llo
Substring using 1 parameter: World*/
```

7.10 substr()

It is like the substring() method.
The difference is, if the second parameter is provided, it takes the first parameter as start Index and second parameter as length for slicing string.

```
let myString = "Hello World";
console.log("Substr using 2 parameters: "+myString.substr(2,5));
console.log("Substr using 1 parameter: "+myString.substr(5));
/*Returns:
Substr using 2 parameters: llo W
Substr using 1 parameter: World*/
```

7.11 toLowerCase()

Converts characters in string to lowercase.

```
let myString = "Hello World";
console.log("Lower case string: "+myString.toLowerCase());
//Returns: Lower case string: hello world
```

7.12 toUpperCase()

Converts characters in string to uppercase

```
let myString = "Hello World";
console.log("Upper case string: "+myString.toUpperCase());
//Returns: Upper case string: HELLO WORLD
```

7.13 trim()

The trim() method removes whitespace from both ends of a string. Whitespace in this context is all the whitespace characters (space, tab, no-break space, etc.) and all the line terminator characters (LF, CR, etc.).

```
const greeting = '   Hello world!   ';
console.log(greeting);
// expected output: "   Hello world!   ";
console.log(greeting.trim());
// expected output: "Hello world!";
```

8. Regular Expression

Regular Expression or regex is basically a sequence of characters indicating a pattern. With the help of this pattern, we can search or match with other strings which follow the pattern indicated.
We will now see how we can validate the name using regular expressions. Let's say that the name must not have $ symbol in it. Here is the implementation of the validateName() function.

```
function validateName(name){
   if(name.match(/\$/)){
      return false;
   }
   else {
      return true;
   }
}
```

The sequence of characters /\$/ is an regular expression. The regular expression used in the above code indicates "any character which is $"
When a regular expression is passed as a parameter to a match function, it checks if the pattern is present in the given string. If found the match function returns an array, else null.

The RegExp object can be constructed using either of the two ways:

- using RegExp constructor
- as a literal value by enclosing within forward-slash (/)

```
let myPattern1 = new RegExp(pattern, modifiers);
let myPattern2 = /pattern/modifiers
/*Here,
pattern specifies the string for regular expression
modifiers are optional
*/
```

Brackets help us define a pattern that enables the search of a given character or a digit in a string or a number.

Pattern	Description
[abc]	To search in a given string for any of the characters present within the brackets
[0-9]	To search in a given string for any of the digits present within the brackets
(a|b)	To search in a given string for either of the characters separated by '|'
[^abc]	To search in a given string for any of the characters which are not a,b, or c.
[^0-9]	To search in a given string for any of the digits which is not between 0-9

Quantifiers help us define a pattern that enables the search of a set of characters or digits in a string or a number.

Pattern	Description
n+	To check if the given string contains at least one “n”.
n*	To check if the given string contains at least zero or more occurrences of n.
n?	To check if the given string contains at least zero or one occurrence of n.
?=n	To match any string that is followed by a specific string n.
n{x}	To match the given string containing X n’s.
n{x,}	To match the given string contain at least X n’s.
n{x,y}	To match the given string containing X to Y n’s.

Predefined classes are set of meta characters grouped together and given a special symbol.

Meta Character	Description	Alternative
\w	Alphanumeric characters and the underscore	[A-Za-z0-9_]
\W	Non-word characters	[^A-Za-z0-9_]
\d	Digits	[0-9]
\D	Non-digits	[^0-9]
\s	Whitespace characters	[\t\n\f\r]
\S	Non whitespace characters	[^\t\n\f\r]

To escape any characters which has special meaning just prefix it by '\' . For example, \$ indicates escape the special meaning of $ and treat it as a regular character.

RegExp object has a very useful method: test().
It offers the simplest way to match the given string with the expected pattern.
It takes a parameter which is the string to be inspected against the expected pattern. When the actual pattern is compared with the expected pattern, this function returns a boolean that is either true or false.
Let us consider the following code that uses RegExp to validate the given email address:

```
let emailPattern = new RegExp("(?=.*@)(?=.+.com)");
let emailString = prompt("Enter email id(someone@xyz.abc)");
if(!(emailPattern.test(emailString))) {
    alert("Email Id is invalid! It should contain@ and .");
}
else {
   alert("Email Id is Valid");
}
```

Additionally, we can use two string methods: search() or replace() to look for the presence of the desired pattern in a given string.

```
let myPattern = /@gmail.com/;
let myString="mark.christ@gmail.com";
console.log(myString.search(myPattern));
console.log(myString.replace(myPattern,'@facebook.com'));
/*
OUTPUT:
11
mark.christ@facebook.com
*/
```

9. Math

It is the JavaScript object that is used to make mathematical calculations on the web.
We can call properties and methods of this object without instantiation of this object because the Math object cannot be instantiated.

Properties:

PI - holds the value of the ratio of the circle's circumference to its diameter.
SQRT2 - holds the value of the square root of 2

```
Math.PI;//Returns 3.14159265358793
Math.SQRT2;//Returns 1.4142135623730951
```

9.1 max()

It accepts multiple numeric values and returns the maximum out of them.

```
Math.max(10,20,20.4,20.6,30.5);
//Returns: 30.5
```

9.2 min()

It accepts multiple numeric values and returns the minimum out of them.

```
Math.min(10,20,20.4,20.6,30.5);
//Returns: 10
```

9.3 ceil()

It returns the upward rounded value of the given number.

```
Math.ceil(20.4);
//Returns: 21
```

9.4 floor()

It returns the downward rounded value of the given number.

```
Math.floor(20.4);
//Returns: 20
```

9.5 random()

It returns any random number between 0 and 1 inclusive of 0 and exclusive of 1.

```
Math.random();
//Returns: 0.19083299074925186
```

9.6 round()

It returns the value of the given number rounded to the nearest integer.

```
Math.round(30.5);
//Returns: 31
```

9.7 sqrt()

It returns square root of given number.

```
Math.sqrt(9);
//Returns: 3
```

10. Functions

Functions are one of the integral components of JavaScript. A JavaScript Function is a set of statements that performs a specific task. They become a reusable unit of code.
In JavaScript, functions are first-class objects. i.e., functions can be passed as an argument to other functions, it can be a return value of another function or can be assigned as a value to a variable. JavaScript leverages this behavior to extend its capabilities.

Functions in JavaScript :

- Can be used to create methods
- Can be used to create classes
- Can be treated as an value
- Can be passed as arguments to another function
- Can inherit from other objects
- Can have user defined properties and methods

10.1 Function Declaration

Developer can declare Function using function keyword followed by user-defined function name along with parentheses().
Based on requirement these parentheses can have zero or more than one parameters separated by commas.

```
function functionName (parameter1, parameter2,…){
 // zero or more than one executable statements
}
```

10.2 Function Expression

It is a concept in which function implementation can be stored into variable. In ES2015 this concept has been implemented through arrow functions.

```
var x= function functionName(parameter1, parameter2,…){
 // zero or more than one executable statements
}
```

10.3 Difference between Function Declaration and Expression

Declaration	Expression
Can define function without holding into variable name. This concept is called as named function declaration	Can declare function holding its implementation into variable name. This concept is called as named function expression Can declare function without any name. This concept is called as Anonymous function
User can develop only standalone implementation which cannot be nested into another non-function code	User can declare this implementation in any other inner function and can be invoked accordingly.
Semicolon is not needed	Semicolon is needed
Implementation can appear only in "program code" i.e.., as a part of another nested function block execution and enclosed variables gets global scope by default	Implementation can appear anywhere in the code and enclosed variables gets local scope by default

10.4 Functions as objects

In JavaScript, functions are actually objects. That means a function can be stored in a variable. For example:

```
funVariable= function myFunc(num1,num2) {
    num3=num1*num2;
    return num3
 }
```

Now, the function is stored inside a variable, we can invoke it using the variable name. For example:

```
console.log(funVariable(10,20));
// 200
```

Since functions are treated as objects you can also pass them as a parameter to another function. For example, in the below code, we are passing the functions welcome() and goodbye() as parameters to the function greet()

```
function welcome(){console.log("Hello World");}
function goodbye(){console.log("See you later");}
function greet(choice){
choice();
}
greet(welcome);
greet(goodbye);
```

Since functions are treated as objects, you can also return them from a function. For example, in the below code, we are returning the function welcome() (stored in variable hello) from a function greet():

```
function greet(){
        var hello=function welcome(){console.log("Hello World");}
        return hello;
}
var retFunc=greet();
retFunc();
```

Higher Order Functions

Functions which can either accept other functions as parameters or return other functions as parameters are called as Higher Order Functions. Many in-built functions in JS are Higher Order Functions.

First Class Citizen:

Any object which can be assigned, passed as a parameter and returned from a function is called a First Class Citizen in a programming language. Thus, all functions are First Class Citizens in JS.

10.5 Arrow Functions

Functions without a name are called anonymous functions. For example:

```
function greet(choice){
        choice();
}
greet(function(){ console.log("Hello World")});
// Hello World
```

An arrow function is a concise way of writing a function. Arrow functions are anonymous functions as they don't have a name.

```
        (parameter) => function body
```

```
let sayHello = () => {
        console.log("Welcome to JavaScript");
};
sayHello();
```

There are two parts for the Arrow function syntax:

- let sayHello = ()
  This declares a variable sayHello and assigns a function to it using () to just say that the variable is actually a function.
- { }: This declares the body of the function with an arrow and the curly braces.

Below are a few scenarios of arrow functions.

Syntax 1: Multi-parameter, multi-line code:

If code is in multiple lines, we need to have {}.

```
calculateCost = (ticketPrice, noOfPerson)=>{
   noOfPerson= ticketPrice * noOfPerson;
   return noOfPerson;
}
console.log(calculateCost(500, 2));
// 1000
```

Syntax 2: No parameter, single line code:

If the code is single line, we don't need {}. The expression is evaluated and automatically returned.

```
trip = () => "Let's go to trip."
console.log(trip());
// Let's go to trip.
```

Syntax 3: One parameter, single line code:

If only one parameter, we don't need ().

```
trip = place => "Trip to " + place;
console.log(trip("Paris"));
// Trip to Paris
```

Syntax 4: One parameter, single line code:

if only one parameter, we can simply use '_' and not use a variable name also.

```
trip = _ => "Trip to " + _;
console.log(trip("Paris"));
// Trip to Paris
```

```
const myObject = {
  items: [1],
  myMethod() {
    console.log(this == myObject) // => true
    this.items.forEach(() => {
      console.log(this === myObject) // => true
      console.log(this === window); // => false
    });
  }
};
myObject.myMethod();
```

Arrow functions do not have their own 'this'. If 'this' is accessed, then its value is taken from the outside of the arrow function. So, in the above-mentioned code, the value of 'this' inside the arrow function equals to the value of 'this' of the outer function, that is, myObject.

10.6 Function Parameters

Function parameters are the variables that are defined in the function definition and the values passed to the function when it is invoked are called arguments.
In JavaScript, function definition does not have any data type specified for the parameters, and type checking is not performed on the arguments passed to the function.
JavaScript does not throw any error if the number of arguments passed during a function invocation doesn't match with the number of parameters listed during the function definition. If the number of parameters is more than the number of arguments, then the parameters that have no corresponding arguments are set to undefined.

```
function multiply(num1, num2) {
        if (num2 == undefined) {
                num2 = 1;
        }
        return num1 * num2;
}
console.log(multiply(5, 6)); // 30
console.log(multiply(5)); // 5
```

JavaScript introduces an option to assign **default values** in functions.

```
function multiply(num1, num2 = 1) {
   return num1 * num2;
}
console.log(multiply(5, 5)); //25
console.log(multiply(10)); //10
console.log(multiply(10, undefined)); //10
```

In the above example, when the function is invoked with two parameters, the default value of num2 will be overridden and considered when the value is omitted while calling.

Rest parameter syntax allows us to hold an indefinite number of arguments in the form of an array.

The rest of the parameters can be included in the function definition by using three dots (…) followed by the name of the array that will hold them.

```
function showNumbers(x, y, …z) {
   return z;
}
console.log(showNumbers(1, 2, 3, 4, 5)); // [3,4,5]
console.log(showNumbers(3, 4, 5, 6, 7, 8, 9, 10)); // [5,6,7,8,9,10]
```

The rest parameter should always be the last parameter in the function definition.

Destructuring gives a syntax which makes it easy to unpack values from arrays, or properties from objects, into different variables.

Array destructuring in functions

```
let myArray = ["Andrew", "James", "Chris"];
function showDetails([arg1, arg2]) {
        console.log(arg1); // Andrew
        console.log(arg2); // James
}
showDetails(myArray);
```

In the above example, the first two array elements ‘Andrew’ and 'James’ have been destructured into individual function parameters arg1 and arg2.

Object destructuring in functions

```
let myObject = { name: "Mark", age: 25, country: "India" };
function showDetails({ name, country }) {
        console.log(name, country); // Mark India
}
showDetails(myObject);
```

The properties name and country of the object have been destructured and captured as a function parameter.

10.7 Built-in functions

JavaScript comes with some built-in functions. To use them, we just need to invoke them.

alert()

It throws an alert box and is often used when user interaction is required to decide whether execution should proceed or not.

```
alert("Let us proceed");
```

confirm()

It throws a confirm box where user can click "OK" or "Cancel". If "OK" is clicked, the function returns "true", else returns "false".

```
let decision = confirm("Shall we proceed?");
```

prompt()

It produces a box where user can enter an input. The user input may be used for some processing later. This function takes parameter of type string which represents the label of the box.

```
let userInput = prompt("Please enter your name:");
```

isNaN()

This function checks if the data-type of given parameter is number or not. If number, it returns "false", else it returns "true".

```
isNaN(30);       //false
isNaN('hello');  //true
```

isFinite()

It determines if the number given as parameter is a finite number. If the parameter value is NaN,positive infinity, or negative infinity, this method will return false, else will return true.

```
isFinite(30);     //true
isFinite('hello'); //false
```

parseInt()

This function parses string and returns an integer number.

It takes two parameters. The first parameter is the string to be parsed. The second parameter represents radix which is an integer between 2 and 36 that represents the numerical system to be used and is optional.

The method stops parsing when it encounters a non-numerical character and returns the gathered number.

It returns NaN when the first non-whitespace character cannot be converted to number

```
parseInt("10");         //10
parseInt("10 20 30");   //10, only the integer part is returned
parseInt("10 years");   //10
parseInt("years 10");   //NaN, the first character stops the parsing
```

parseFloat()

This function parses string and returns a float number.

The method stops parsing when it encounters a non-numerical character and further characters are ignored.

It returns NaN when the first non-whitespace character cannot be converted to number.

```
parseFloat("10.34");      //10.34
parseFloat("10 20 30");   //10
parseFloat("10.50 years"); //10.50
```

eval()

It takes an argument of type string which can be an expression, statement or sequence of statements and evaluates them.

```
eval("let num1=2; let num2=3;let result= num1 * num2;console.log(result)");
```

JavaScript provides timer built-in functions. Let us explore these timer functions.

setTimeout()

It executes a given function after waiting for the specified number of milliseconds.
It takes 2 parameters. First is the function to be executed and the second is the number of milliseconds after which the given function should be executed.

```
function executeMe(){
console.log("Function says hello!")
}
setTimeout(executeMe, 3000);
//It executes executeMe() after 3 seconds.
```

clearTimeout()

It cancels a timeout previously established by calling setTimeout().
It takes the parameter "timeoutID" which is the identifier of the timeout that can be used to cancel the execution of setTimeout(). The ID is returned by the setTimeout().

```
function executeMe(){
console.log("Function says hello!")
}
let timerId= setTimeout(executeMe, 3000);
clearTimeout(timerId);
```

setInterval()

It executes the given function repetitively.
It takes 2 parameters, first is the function to be executed and second is the number of milliseconds. The function executes continuously after every given number of milliseconds.

```
function executeMe(){
console.log("Function says hello!");
}
setInterval(executeMe,3000);
//It executes executeMe() every 3 seconds
```

clearInterval()

It cancels the timed, repeating execution which was previously established by a call to setInterval().
It takes the parameter "intervalID" which is the identifier of the timeout that can be used to cancel the execution of setInterval(). The ID is returned by the setInterval().

```
function executeMe(){
console.log("Function says hello!");
}
let timerId=setInterval(executeMe, 2000);
function stopInterval(){
        clearInterval(timerId);
        console.log("Function says bye to setInterval()!"}
setTimeout(stopInterval,5000)
//It executes executeMe() every 2 seconds and after 5 seconds, further calls to executeMe() is stopped.
```

10.8IIFE (Immediate Involving Function Expression)

IIFE (Immediate Involving Function Expression) represents JavaScript function without any name, which would be executed as soon as interpreter runs that block. This concept can also be referred as self-executing anonymous function.
Developer can avoided variable hoisting concept, whereas interpreter provides public access to the declared method and internal maintains privacy also.

```
(function() {
  // zero or more than one executable statements along with return statement.
})();
```

10.9Closure

Is a concept through which developer can implement lexical scoping through functions.

Closure internal are built on below mentioned principles :

- Inner/Nested functions
- Function who can return another function.

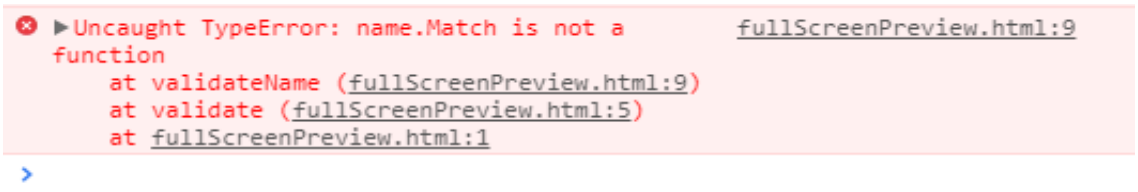
In short you can consider Closure as a inner function which can outlive the lifetime of the outer function and hence can continue to access the variables of the outer function.
In the below example, the outer function is returning the inner function. Thus we see that we are able to invoke the inner function many times without having to invoke outer function each time.

```
function outer(){
console.log("Outer")
        function inner(){
                return "Inner"
        }
return inner
}
```

```
private_function=outer()
private_function()
private_function()
private_function()
private_function()
```

11. Error Handling

When an error occurs, the stack trace of the error is displayed in the console.
It provides information regarding the name, message, error stack, and the location of the error, which helps in debugging.



In JavaScript, all errors are of type error object. These objects carry the information related to the error, including the stack trace.
Whenever an exceptional event occurs, the browser environment generates the error object and throws it.
The moment an error object is thrown, further execution of program is stopped. If it error is not handled, then the error will be propagated to the calling environment.
The calling environment can be either a calling method, or the browser.

```
function validateName(name){
   if(name.Match(/[\$\#]/)){
      console.log("Input is invalid");
      return false;
   }
   else {
      console.log("Input is valid");
      return true;
   }
}
function validate(){
validateName("Hello");
}
validate()
```

Exception propagation steps:

- In the above scenario, an error is generated inside validateName() method
- Since validateName() method doesn't handle it, the error is propagated back to its calling environment, which is the validate()
- Since validate() method doesn't handle it, the error is propagated back to its calling environment, which is button click event
- Since the error is not even handled in button click event, it is propagated back to its calling environment i.e. browser

When the browser receives the error, it shows the stack trace in the console and terminates the program.
Hence, the error started from validateName(), propagated to validate() and then propagated back to the browser.

As we saw in the console, the Error object contains three properties:

- name: defines name/type of the error. For this example the error name is 'TypeError'.
- message: is a short description about the error. In our case, it is 'name.Match is not a function', because Match() function with uppercase 'M' does not exist.
- stack: A full stack trace of the error, with error name, error message, file name, method, line information about where the error has occurred. Here, the last three lines starting with "at" is the stack trace.

There can be different values for the name property, which signify different Error objects thrown during the execution of JS program.

Here are some built-in error objects in JS

- EvalError: Is an instance of Error which represents than an error occurred regarding the global function. Example: eval().
- InternalError: Is an instance of Error which represents an internal error in the JavaScript engine. Example : "too much recursion".
- RangeError:Is an instance of Error which represents than an error occurred when a numeric variable or parameter is outside of its valid range.
- ReferenceError: Is an instance of Error which represents than an error occurred when de-referencing an invalid reference.
- SyntaxError: Is an instance of Error which represents than an error occurred while parsing some input in eval() or in JSON.parse().
- TypeError: Is an instance of Error which represents than an error occurred when a variable or parameter is not of a valid type.
- URIError: Is an instance of Error which represents than an error occurred when encodeURI() or decodeURI() are passed with invalid parameters.

11.1 try-catch block

Error handling is important, as unhandled errors can lead to abrupt termination of the program. These errors can be handled by using try-catch block.

```
function validateName(name) {
      try {
         if (name.Match(/[\$\#]/)) {// error occurs here
            /* All the below lines of try do not run
               as error was thrown in previous line*/
            return false;
         }
         else {
            return true;
         }
      }
      catch (error) {
         // code for Handling error
         console.log(error.message);
      }
   }
validateName("Josh")
```

The code that can throw an error should be enclosed inside the try block. A try block should be immediately followed by a catch block.
A catch block is an error handler which can handle the error.
The error object thrown from try block will be passed as parameter to catch block.
In the above case, when validateName() throws an error because of wrong method name, the error object is created and thrown. This error object is caught by the catch block and performs appropriate handling of error, here we have just logged the message present in the error object.
Once the error object has been thrown, the next immediate lines in the try block will not be executed.

Now that we have seen different types of Errors, can we handle each of these errors in separate manner? The answer is no. This is because JavaScript is dynamically typed language, so we cannot specify the different catch blocks for each error instance.

Instead, which we can use some conditional statements inside the catch block.

```
function validateName(name) {
      try {
         if (name.Match(/[\$\#]/)) {// error occurs here
            /* All the below lines of try do not run
               as error was thrown in previous line*/
            return false;
         }
         else {
            return true;
         }
      }
      catch (error) {
         if (error instanceof TypeError)
            console.log("Type Error Occurred");
         else if (error instanceof RangeError)
            console.log("Range Error Occurred");
         else if (error instanceof SyntaxError)
            console.log("Syntax Error Occurred");
         else
            console.log("Some other Error Occurred");
      }
   }
validateName("Josh$");
```

Here, once an error is thrown in the try block, it will be handled by the catch block. Inside the catch block, we are checking the type of the error object by using instanceof operator and handling them separately.

11.2 finally block

An error inside a try block causes the rest of the code to be skipped. This might lead to some important parts of the code not being executed.
There may be some important code which must be executed in all the conditions.

For example:

- Closing the database or file connection
- Releasing the memory allocated for objects

Hence, keeping these such code inside the try block cannot guarantee their execution.

In such situations, the finally block plays an important role. The finally block ensures that the code will be executed, irrespective of whether an error has occurred or not.

```
function validateName(name) {
      try {
         if (name.Match(/[\$\#]/)) {// error occurs here
            /* All the below lines of try do not run
               as error was thrown in previous line*/
            return false;
         }
         else {
            return true;
         }
      }
      catch (error) {
            console.log("Error Occurred");
      }
      finally{
         console.log("Cleaning up resources");
      }
   }
validateName("josh$");
```

Note: A try block should be always followed by either a catch block or a finally block or both.

10.10 Nested Functions

In JavaScript, it is perfectly normal to have functions inside functions. The function within another function body is called a nested function.
The nested function is private to the container function and cannot be invoked from outside the container function.
Internally Nested Function would be private to their Containing Function. Due which developers cannot access directly the argument and variables of Inner Function. To access either of them, you need to invoke your logic with the help of Outer Function only and vice-versa Outer Function cannot access directly Inner Functions declared variable and arguments, to access them developer has to execute statements in the Containing Function.

```
function giveMessage(message) {
        let userMsg = message;
        function toUser(userName) {
                let name = userName;
                let greet = userMsg + " " + name;
                return greet;
        }
        userMsg = toUser("Bob");
        return userMsg;
}
console.log(giveMessage("The world says hello dear: "));
// The world says hello dear: Bob
```

11.3 throw

Apart from the code throwing errors, we can also programmatically create our own errors and throw them to change the flow of execution.
This can be done by creating a new object of Error class and passing our own name and message to it.

```
var err = new Error();      //You can pass the message or not it is optional
err.name = "InvalidEmailError";
err.message ="Invalid Email";
throw err;
```

```
function validateName(name) {
  try {
        if (name.match(/\$/)) {
        throw new Error("Name should not contain $");
        }
   else {
            return true;
        }
}
 catch (error) {
        console.log(error.message);
}
finally{
```

```
console.log("Cleaning up resources");
}
}
validateName("Hello$");
```

- When an error occurs, the JavaScript program crashes and we need to handle it
- Exceptions propagate from across functions until handled
- try, catch, finally is used to handle errors
- finally block of code will always run

12. Objects

Objects plays very vital role in JavaScript programming.
They can represent any real-world thing and it is constructs with properties and methods.

12.1 Creating Object using Literal notation

Objects can be created using object literal notation. Object literal notation is a comma-separated list of name-value pairs wrapped inside curly braces. This promotes the encapsulation of data in a tidy package. This is how the objects in JavaScript are created using the literal notation:

```
objectName = {
    //-------------states of the object-----------
    key_1: value_1,
    key_2: value_2,
    ...
    key_n: value_n,
    //-------------behaviour of the object---------
    key_function_name_1: function (parameter) {
        //we can modify any of the property declared above
    },
    ...
    key_function_name_n: function(parameter) {
        //we can modify any of the property declared above
    }
}
```

12.2 Creating Object using Enhanced Object Literals

Below is the older syntax used to create object literals.

```
let name = "Arnold";
let age = 65;
let country = "USA";
let obj = {
    name: name,
    age: age,
    country: country
};
```

The modern way to create objects in a simpler way :

```
let name="Arnold";
let age=65;
let country="USA";
let obj={name,age,country};
```

12.3 Creating Object using Enhanced Object Literals - Computed Property

Earlier in JavaScript to add a dynamic property to an existing object below syntax is used.

```
let personalDetails = {
    name: "Stian Kirkeberg",
    country: "Norway"
};
let dynamicProperty = "age";
personalDetails[dynamicProperty] = 45;
console.log(personalDetails.age);  //Output: 45
```

With newer updates in JavaScript after 2015 the dynamic properties can be conveniently added using hash notation and the values are computed to form a key-value pair.

```
let dynamicProperty = "age";
let personalDetails = {
    name: "Stian Kirkeberg",
    country: "Norway",
    [dynamicProperty]: 45
};
console.log(personalDetails.age);   //Output: 45
```

12.4 Combining and Cloning of Objects using Spread Operator

We can make use of the spread operator to combine two or more objects created. The newly created object will hold all the properties of the merged objects.

```
let object1Name = {
   //properties
};
let object2Name = {
   //properties
};
let combinedObjectName = {
   ...object1Name,
   ...object2Name
};
//the combined object will have all the properties of object1 and object2
```

It is possible to get a copy of an existing object with the help of the spread operator.

```
let originalObj = { one: 1, two: 2, three: 3 };
let clonedObj = { ...originalObj };
/*
Here spreading the object into a list of parameters happens
which return the result as a new object
checking whether the objects hold the same contents or not
*/
alert(JSON.stringify(originalObj) === JSON.stringify(clonedObj)); // true
//checking whether both the objects are equal
alert(originalObj === clonedObj); // false (not same reference)
//to show that modifying the original object does not alter the copy made
originalObj.four = 4;
alert(JSON.stringify(originalObj)); // {"one":1,"two":2,"three":3,"four":4}
alert(JSON.stringify(clonedObj)); // {"one":1,"two":2,"three":3}
```

Object.assign()

The Object.assign() method copies all enumerable own properties from one or more source objects to a target object. It returns the modified target object.

```
Object.assign(target, source);
```

For deep cloning, we need to use alternatives, because Object.assign() copies property values.
If the source value is a reference to an object, it only copies the reference value.

```
function test() {
  'use strict';
  let obj1 = { a: 0 , b: { c: 0}};
  let obj2 = Object.assign({}, obj1);
  console.log(JSON.stringify(obj2)); // { "a": 0, "b": { "c": 0}}

  obj1.a = 1;
  console.log(JSON.stringify(obj1)); // { "a": 1, "b": { "c": 0}}
  console.log(JSON.stringify(obj2)); // { "a": 0, "b": { "c": 0}}

  obj2.a = 2;
  console.log(JSON.stringify(obj1)); // { "a": 1, "b": { "c": 0}}
  console.log(JSON.stringify(obj2)); // { "a": 2, "b": { "c": 0}}

  obj2.b.c = 3;
  console.log(JSON.stringify(obj1)); // { "a": 1, "b": { "c": 3}}
  console.log(JSON.stringify(obj2)); // { "a": 2, "b": { "c": 3}}

  // Deep Clone
  obj1 = { a: 0 , b: { c: 0}};
  let obj3 = JSON.parse(JSON.stringify(obj1));
  obj1.a = 4;
  obj1.b.c = 4;
  console.log(JSON.stringify(obj3)); // { "a": 0, "b": { "c": 0}}
}test();
```

12.5 Destructuring objects

Destructuring gives a syntax which makes it easy to create objects based on variables.
It also helps to extract data from an object. Destructuring works even with the rest and spread operators.
In the below example an object is destructured into individual variables:

```
let myObject = { name: 'Arnold', age: 65, country: 'USA' };
let { name, age:currentAge } = myObject; //alias can be used with :
console.log(name);
console.log(currentAge);
```

Object destructuring in functions

```
let myObject = { name: 'Marty', age: 65, country: 'California' };
function showDetails({ country }) {
  console.log(country);
}
showDetails(myObject);  //invoke the function using the object
//OUTPUT: California
```

12.6 Accessing object properties

Once the object has been created, its variables or methods can be accessed in 2 different ways:

Using

- dot operator
  objectName.key;

- bracket operator
  objectName[key];

  this access should be used mainly when the property names are having space, hyphen, or one that starts with a number.

12.7 Iterating an object

```
var empOne = {
   name : "John",
   empNumber : 1001,
   emailId : "John@gmail.com"
};
```

Some of the ways of iterating over this object are:

for..in:

The for..in loop iterates over the object and gives the property values of the object. For example:

```
for(let property in empOne){
  console.log(empOne[property]);
}
```

Object.values():

Object.values() will give all the values of an object in an array. For example:

```
console.log(Object.values(empOne));
```

Object.keys():

The **Object.keys()** method returns an array of a given object's own enumerable property **names**, iterated in the same order that a normal loop would.

```
Object.keys(obj)
```

Object.entries():

The **Object.entries()** method returns an array of a given object's own enumerable string-keyed property [key, value] pairs

```
const object1 = {
  a: 'somestring',
  b: 42
};

for (const [key, value] of Object.entries(object1)) {
  console.log(`${key}: ${value}`);
}

// expected output:
// "a: somestring"
// "b: 42"
```

12.8 Delete property

The JavaScript delete operator removes a property from an object; if no more references to the same property are held, it is eventually released automatically.

```
delete object.property

delete object[‘property’]
```

12.9 Getters and Setters

The get syntax binds an object property to a function that will be called when that property is looked up.

```
{get prop() { ... } }

{get [expression]() { ... } }
```

The set syntax binds an object property to a function to be called when there is an attempt to set that property.

```
{set prop(val) { . . . }}

{set [expression](val) { . . . }}
```

```
const person = {
  firstName: "John",
  lastName: "Doe",
  language: "",
  set lang(lang) {
    this.language = lang;
  }
get lang() {
    return this.language;
  }
};

// Set an object property using a setter:
person.lang = "en";


// Display data from the object using a getter:
document.getElementById("demo").innerHTML = person.lang;
```

12.10 Method

Functions can be used to create methods in a object literal concept.
Behavior of an object can be represented through methods.

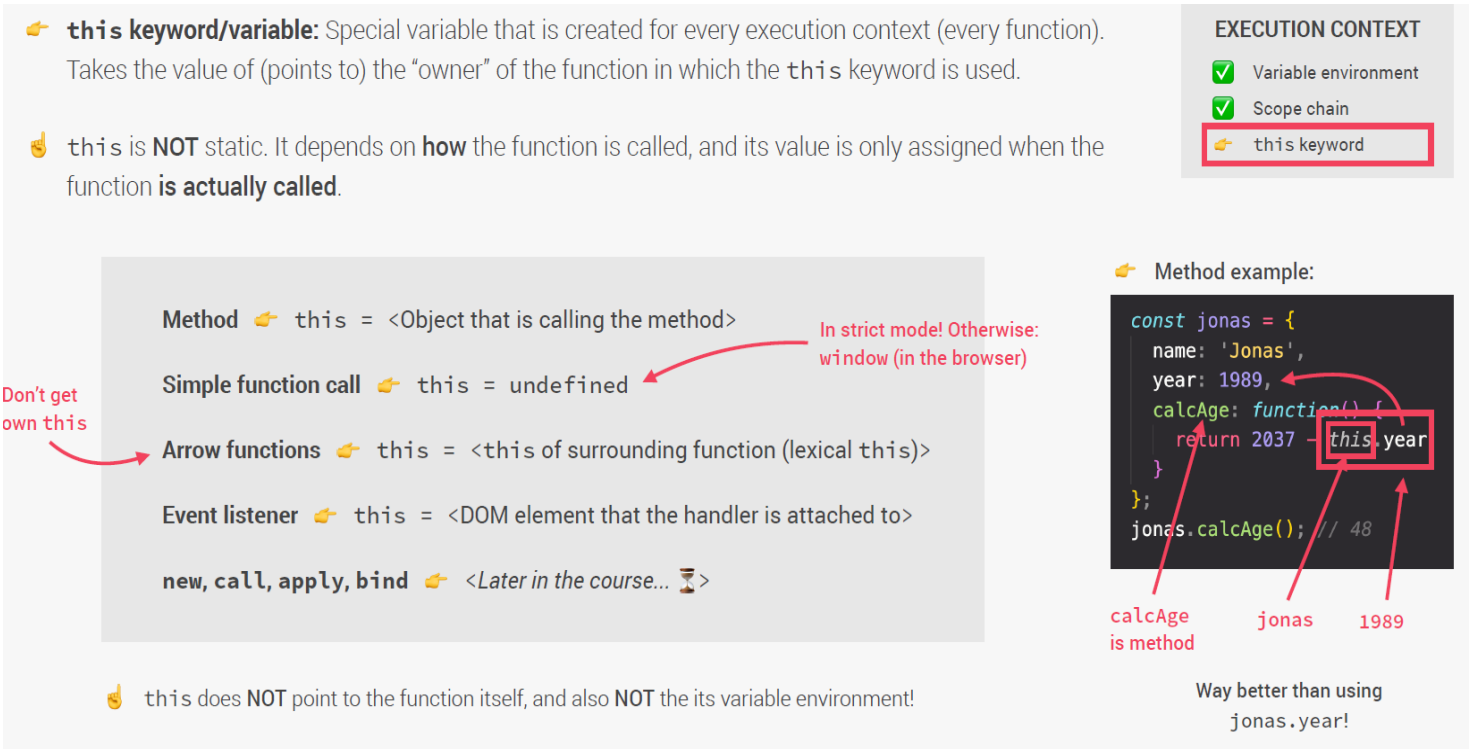
```
var doctor_obj = {
    name: "Tom",
    specialization: "ENT",
    availability: function() {
        return“ From Morning 9 A.M - 11 A.M "}
    }
    console.log(doctor_obj.name);
    console.log(doctor_obj.availability());
}
```

12.11 this keyword

It is one of the special keywords available in JavaScript. Which can holds a reference variable of the current object on which the method is being executed.

Using this keyword, developer can access the properties inside a method.

In most cases, the value of this is determined by how a function is called (runtime binding).



12.12 bind, call and apply

The bind() method creates a new function that, when called, has its this keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

```
bind(thisArg)

bind(thisArg, arg1, ... , argN)
```

```
var myButton = {
  content: 'OK',
  click() {
    console.log(this.content + ' clicked');
  }
};

myButton.click();
```

```
var looseClick = myButton.click;
looseClick(); // not bound, 'this' is not myButton - it is the globalThis

var boundClick = myButton.click.bind(myButton);
boundClick(); // bound, 'this' is myButton
```

The call() method calls a function with a given this value and arguments provided individually.

```
call(thisArg)
call(thisArg, arg1, ... , argN)
```

```
const person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName + "," + city + "," + country;
  }
}

const person1 = {
  firstName:"John",
  lastName: "Doe"
}

person.fullName.call(person1, "Oslo", "Norway");
```

The apply() method calls a function with a given this value, and arguments provided as an array (or an array-like object).

```
apply(thisArg)
apply(thisArg, argsArray)
```

```
const person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName + "," + city + "," + country;
  }
}
const person1 = {
  firstName:"John",
  lastName: "Doe"
}
person.fullName.apply(person1, ["Oslo", "Norway"]);
```

Difference between call() and apply() method:

The only difference is **call()** method takes the arguments separated by comma while **apply()** method takes the array of arguments.

13. JSON

JSON an acronym for JavaScript object notation.
It is a lightweight data-interchange format used for storing and sharing data between client and server over the network.
For example, to store and share customer information over the web, this is how the corresponding JSON data will look like :

```
let data = {
   "customers": [
      { "firstName": "Bob", "lastName": "Morry" },
      { "firstName": "Albert", "lastName": "Smith" },
      { "firstName": "Kate", "lastName": "Ward" }
   ]
};
//Where data is the JSON object and
//customers is the array name
```

In this code, the variable 'data' is exactly like the literal notation syntax used for object creation in JavaScript. Whereas there is a very small difference.
For JavaScript objects we do not put the key in quotes and if values are of string data type they can be put in single or double-quotes.
But for JSON object, it is mandatory to put the key inside the double quotes and all the values of type string inside the double-quotes.
JSON is a text-only format. It travels over the network as a string.

```
JavaScript Object: { firstName: "Sam", lastName: "Fernandes" }
//key need not be enclosed within quotes for JavaScript Objects
JSON Object: { "firstName": "Sam", "lastName": "Fernandes" }
//key must be enclosed within quotes for JSON Objects
```

JSON – Methods

parse()

Used to parse a string as JSON and helps the program to process objects.

```
let stringJSON = '{"firstName":"Sam","lastName":"Fernandes"}'
let obj = JSON.parse(stringJSON);
console.log(obj);
//OUTPUT: { firstName: 'Sam', lastName: 'Fernandes' }
```

stringify()

Returns the JSON string corresponding to the given object.

```
let dataJSON = { firstName: "Sam", lastName: "Fernandes" };
let obj = JSON.stringify(dataJSON);
console.log(obj);
//OUTPUT: {"firstName":"Sam","lastName":"Fernandes"}
```

14. Date

The built-in JavaScript object 'Date' allows us to work with dates and times displayed as part of the web page. It can be instantiated wherever required using one of the many constructors available.

```
let dateObject1 = new Date();
console.log("Date is: " + dateObject1);
//OUTPUT: Date is: Thu Jun 18, 2020, 22:17:36 GMT+0530 (India Standard Time)
```

```
let  dataObject2 = new Date(2020, 5, 18, 22, 20, 23, 0000);
console.log("Date is: "+dataObject2);
//OUTPUT: Date is: Thu Jun 18, 2020, 22:20:23 GMT+0530 (India Standard Time)
```

Date - Getter Methods

Method	Description
getDate()	Return the numeric day of the month. The value ranges from 1 to 31.
getDay()	Returns numeric day of week. Value ranges from 0 to 6.
getFullYear()	Return four digit year (YYYY).
getHours()	Returns numeric hour. Value ranges from 0 to 23.
getMonth()	Returns numeric month. Value ranges from 0 to 11.
getMilliseconds()	Returns numeric milliseconds. Value ranges from 0 to 999.
getTime()	Returns number of milliseconds since 1/1/1970 at 12 a.m.

```
let dateObject1 = new Date();
console.log("Date is: " + dateObject1.getDate());
console.log("Day is: " + dateObject1.getDay());
console.log("Year is: " + dateObject1.getYear());
console.log("Horrs: " + dateObject1.getHours());
console.log("Month is: " + dateObject1.getMonth());
console.log("Time is: " + dateObject1.getTime());
console.log("Millisecond: " + dateObject1.getMilliseconds());
/*
OUTPUT:
Date is: 18
Day is: 4
Year is: 120
Horrs: 22
Month is: 5
Time is: 1592499518512
Millisecond: 512
*/
```

Date - Setter Methods

Method	Description
setDate()	Sets the numeric day of the month. Value range from 1 to 31.
setFullYear()	Sets four-digit year (YYYY).
setHours()	Sets numeric hour. The value ranges from 0 to 23.
setMonth()	Sets numeric month. The value ranges from 0 to 11.
setMilliseconds()	Sets numeric milliseconds. The value ranges from 0 to 999.
setTime()	Sets the number of milliseconds from 1/1/1970 at 12 a.m.

```
let dateObject1 = new Date();
dateObject1.setDate(3);
dateObject1.setYear(1996);
dateObject1.setHours(8);
dateObject1.setMonth(7);
dateObject1.setMilliseconds(2000);

console.log("Date is: " + dateObject1.getDate());
console.log("Year is: " + dateObject1.getYear());
console.log("Hours: " + dateObject1.getHours());
console.log("Month is: " + dateObject1.getMonth());
console.log("Millisecond: " + dateObject1.getMilliseconds());
/*
OUTPUT:
Date is: 3
Year is: 96
Hours: 8
Month is: 7
Millisecond: 0
*/
```

The toLocaleString() method returns a string with a language sensitive representation of this date.
The new locales and options arguments let applications specify the language whose formatting conventions should be used and customize the behavior of the function.

```
toLocaleString()
toLocaleString(locales)
toLocaleString(locales, options)
```

```
let date = new Date(Date.UTC(2012, 11, 12, 3, 0, 0));
// toLocaleString() without arguments depends on the
// implementation, the default locale, and the default time zone
console.log(date.toLocaleString());
// → "12/11/2012, 7:00:00 PM" if run in en-US locale with time zone America/Los_Angeles
```

```
let date = new Date(Date.UTC(2012, 11, 20, 3, 0, 0));

// Request a weekday along with a long date
let options = { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' };

console.log(date.toLocaleString('de-DE', options));
// → "Donnerstag, 20. Dezember 2012"

// An application may want to use UTC and make that visible
options.timeZone = 'UTC';
options.timeZoneName = 'short';

console.log(date.toLocaleString('en-US', options));
// → "Thursday, December 20, 2012, GMT"

// Sometimes even the US needs 24-hour time
console.log(date.toLocaleString('en-US', { hour12: false }));
// → "12/19/2012, 19:00:00"
```

The toLocaleDateString() method returns a string with a language sensitive representation of the date portion of this date.

The new locales and options arguments let applications specify the language whose formatting conventions should be used and allow to customize the behavior of the function.

```
const event = new Date(Date.UTC(2012, 11, 20, 3, 0, 0));
const options = { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' };

console.log(event.toLocaleDateString('de-DE', options));
// expected output: Donnerstag, 20. Dezember 2012

console.log(event.toLocaleDateString('ar-EG', options));
// expected output: الخميس، ٢٠ ديسمبر، ٢٠١٢

console.log(event.toLocaleDateString(undefined, options));
// expected output: Thursday, December 20, 2012 (varies according to default locale)
```

The toLocaleTimeString() method returns a string with a language-sensitive representation of the time portion of the date. The newer locales and options arguments let applications specify the language formatting conventions to use. These arguments can also customize the behavior of the function.

```
var date = new Date(Date.UTC(2012, 11, 20, 3, 0, 0));

// an application may want to use UTC and make that visible
var options = { timeZone: 'UTC', timeZoneName: 'short' };
console.log(date.toLocaleTimeString('en-US', options));
// → "3:00:00 AM GMT"

// sometimes even the US needs 24-hour time
console.log(date.toLocaleTimeString('en-US', { hour12: false }));
// → "19:00:00"

// show only hours and minutes, use options with the default locale - use an empty array
console.log(date.toLocaleTimeString([], { hour: '2-digit', minute: '2-digit' }));
// → "20:01"
```

15. DOM

DOM stands for Document Object Model. It models the HTML document into an object. That means, everything we see in the HTML page is represented as objects in JavaScript.
DOM is not a programming language, it is an interface through which JavaScript access the elements of HTML pages.
The DOM represents a document with a logical tree. Each branch of the tree ends in a node, and each node contains objects. DOM methods allow programmatic access to the tree. With them, you can change the document's structure, style, or content.
Nodes can also have event handlers attached to them. Once an event is triggered, the event handlers get executed.

15.1 DOM Methods

Query Methods

15.1.1 querySelector()

The Document method querySelector() returns the first Element within the document that matches the specified selector, or group of selectors. If no matches are found, null is returned.

```
element = document.querySelector(selectors);
```

15.1.2 querySelectorAll()

The Document method querySelectorAll() returns a static (not live) NodeList representing a list of the document's elements that match the specified group of selectors.

```
elementList = document.querySelectorAll(selectors);
```

15.1.3 getElementById()

The Document method getElementById() returns an Element object representing the element whose id property matches the specified string. Since element IDs are required to be unique if specified, they're a useful way to get access to a specific element quickly.

```
var element = document.getElementById(id);
```

15.1.4 getElementByClassName()

The getElementsByClassName method of Document interface returns an array-like object of all child elements which have all of the given class name(s).(live HTML collection)
When called on the document object, the complete document is searched, including the root node. You may also call getElementsByClassName() on any element; it will return only elements which are descendants of the specified root element with the given class name(s).

```
var elements = document.getElementsByClassName(names);
```

15.1.5 getElementByClassName()

The getElementsByTagName method of Document interface returns an HTMLCollection of elements with the given tag name.

The complete document is searched, including the root node. The returned HTMLCollection is live, meaning that it updates itself automatically to stay in sync with the DOM tree without having to call

```
var elements = document.getElementsByTagName(name);
```

Creating & Inserting Elements

15.1.6 Element.insertAdjacentHTML()

The insertAdjacentHTML() method of the Element interface parses the specified text as HTML or XML and inserts the resulting nodes into the DOM tree at a specified position. It does not reparse the element it is being used on, and thus it does not corrupt the existing elements inside that element.

This avoids the extra step of serialization, making it much faster than direct innerHTML manipulation.

```
element.insertAdjacentHTML(position, text);
```

Parameters

position

A DOMString representing the position relative to the element; must be one of the following strings:
'beforebegin': Before the element itself.
'afterbegin': Just inside the element, before its first child.
'beforeend': Just inside the element, after its last child.
'afterend': After the element itself.

text

The string to be parsed as HTML or XML and inserted into the tree.

15.1.7 Document.createElement()

In an HTML document, the document.createElement() method creates the HTML element specified by tagName, or an HTMLUnknownElement if tagName isn't recognized.

```
let element = document.createElement(tagName[, options]);
```

15.1.8 Element.append()

The Element.append() method inserts a set of Node objects or DOMString objects after the last child of the Element. DOMString objects are inserted as equivalent Text nodes.

Differences from Node.appendChild():

- Element.append() allows you to also append DOMString objects, whereas Node.appendChild() only accepts Node objects.
- Element.append() has no return value, whereas Node.appendChild() returns the appended Node object.
- Element.append() can append several nodes and strings, whereas Node.appendChild() can only append one node.

15.1.9 Element.prepend()

The Element.prepend() method inserts a set of Node objects or DOMString objects before the first child of the Element. DOMString objects are inserted as equivalent Text nodes.

15.1.10 Element.before()

The Element.before() method inserts a set of Node or DOMString objects in the children list of this Element's parent, just before this Element. DOMString objects are inserted as equivalent Text nodes.

15.1.11 Element.after()

The Element.after() method inserts a set of Node or DOMString objects in the children list of the Element's parent, just after the Element. DOMString objects are inserted as equivalent Text nodes.

15.1.12 Element.replaceWith()

The Element.replaceWith() method replaces this Element in the children list of its parent with a set of Node or DOMString objects. DOMString objects are inserted as equivalent Text nodes.

Clone Elements

15.1.13 Node.cloneNode()

The Node.cloneNode() method returns a duplicate of the node on which this method was called.

```
let newClone = node.cloneNode([deep])
```

deep Optional

If true, then node and its whole subtree—including text that may be in child Text nodes—is also copied.
If false, only node will be cloned. Any text that node contains is not cloned, either (since text is contained by one or more child Text nodes).
deep has no effect on empty elements (such as the <img> and <input> elements).

Removing Elements

15.1.14 Element.remove()

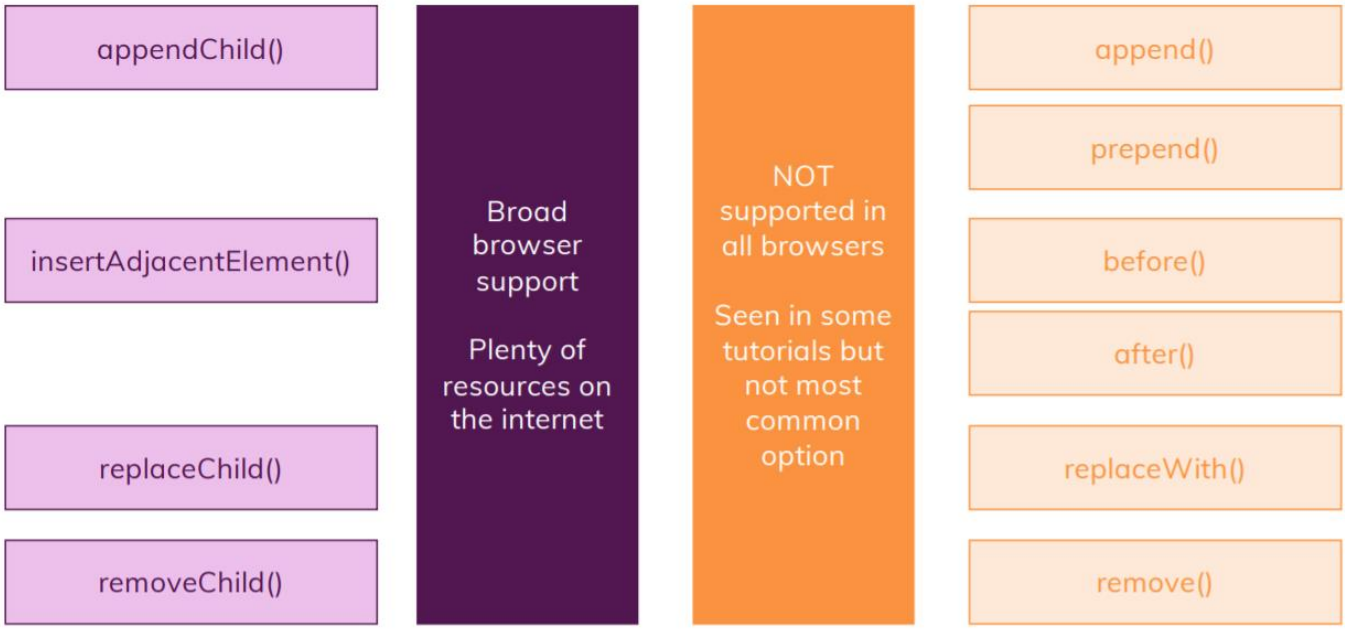
The Element.remove() method removes the element from the tree it belongs to.

15.1.15 Element.removeChild()

The Node.removeChild() method removes a child node from the DOM and returns the removed node.

node.removeChild(child);

- child is the child node to be removed from the DOM.
- node is the parent node of child.



15.2 DOM Properties

15.2.1 Element.children

The read-only children property returns a live HTMLCollection which contains all of the child elements of the element upon which it was called.

Element.children includes only element nodes. To get all child nodes, including non-element nodes like text and comment nodes, use Node.childNodes.

15.2.2 Element.firstElementChild

The Element.firstElementChild read-only property returns an element's first child Element, or null if there are no child elements.

Element.firstElementChild includes only element nodes. To get all child nodes, including non-element nodes like text and comment nodes, use Node.firstChild.

15.2.3 Element.lastElementChild

The Element.lastElementChild read-only property returns an element's last child Element, or null if there are no child elements.

Element.lastElementChild includes only element nodes. To get all child nodes, including non-element nodes like text and comment nodes, use Node.lastChild.

15.2.4 Node.parentElement

The Node.parentElement read-only property returns the DOM node's parent Element, or null if the node either has no parent, or its parent isn't a DOM Element.

To include non-element nodes like text and comment nodes, use Node.parentNode.

15.2.5 Element.closest()

The closest() method traverses the Element and its parents (heading toward the document root) until it finds a node that matches the provided selector string. Will return itself or the matching ancestor. If no such element exists, it returns null. Closest Ancestor.

15.2.6 Node.firstElementChild

The Element.firstElementChild read-only property returns an element's first child Element, or null if there are no child elements

Element.firstElementChild includes only element nodes. To get all child nodes, including non-element nodes like text and comment nodes, use Node.firstChild.

15.2.7 Node.lastElementChild

The Element.lastElementChild read-only property returns an element's last child Element, or null if there are no child elements.

Element.lastElementChild includes only element nodes. To get all child nodes, including non-element nodes like text and comment nodes, use Node.lastChild.

15.2.8 Element.nextElementSibling

The Element.nextElementSibling read-only property returns the element immediately following the specified one in its parent's children list, or null if the specified element is the last one in the list.

To include non-element nodes like text and comment nodes, use Node.nextSibling.

15.2.9 Element.previousElementSibling

The Element.previousElementSibling read-only property returns the Element immediately prior to the specified one in its parent's children list, or null if the specified element is the first one in the list.

To include non-element nodes like text and comment nodes, use Node.previousSibling.

15.2.10 Node.textContent

The textContent property of the Node interface represents the text content of the node and its descendants.

15.2.11 Node.innerHTML

The Element property innerHTML gets or sets the HTML or XML markup contained within the element.

15.2.12 Node.innerText

The innerText property of the HTMLElement interface represents the "rendered" text content of a node and its descendants.

15.2.13 Element.className

The className property of the Element interface gets and sets the value of the class attribute of the specified element.

15.2.14 Element.classList

The Element.classList is a read-only property that returns a live DOMTokenList collection of the class attributes of the element. This can then be used to manipulate the class list.
Using classList is a convenient alternative to accessing an element's list of classes as a space-delimited string via element.className.
Although the classList property itself is read-only, you can modify its associated DOMTokenList using the add(), remove(), replace(), and toggle() methods.

15.2.15 Element.class.cssproperty

The style read-only property returns the inline style of an element in the form of a CSSStyleDeclaration object that contains a list of all styles properties for that element with values assigned for the attributes that are defined in the element's inline style attribute.

While this property is considered read-only, it is possible to set an inline style by assigning a string directly to the style property. In this case the string is forwarded to CSSStyleDeclaration.cssText. Using style in this manner will completely overwrite all inline styles on the element.

Therefore, to add specific styles to an element without altering other style values, it is generally preferable to set individual properties on the CSSStyleDeclaration object. For example, element.style.backgroundColor = "red".

A style declaration is reset by setting it to null or an empty string, e.g., elt.style.color = null.

15.2.16 Element.dataset.*

The dataset read-only property of the HTMLElement interface provides read/write access to custom data attributes (data-*) on elements. It exposes a map of strings (DOMStringMap) with an entry for each data-* attribute.

An HTML data-* attribute and its corresponding DOM dataset.property.

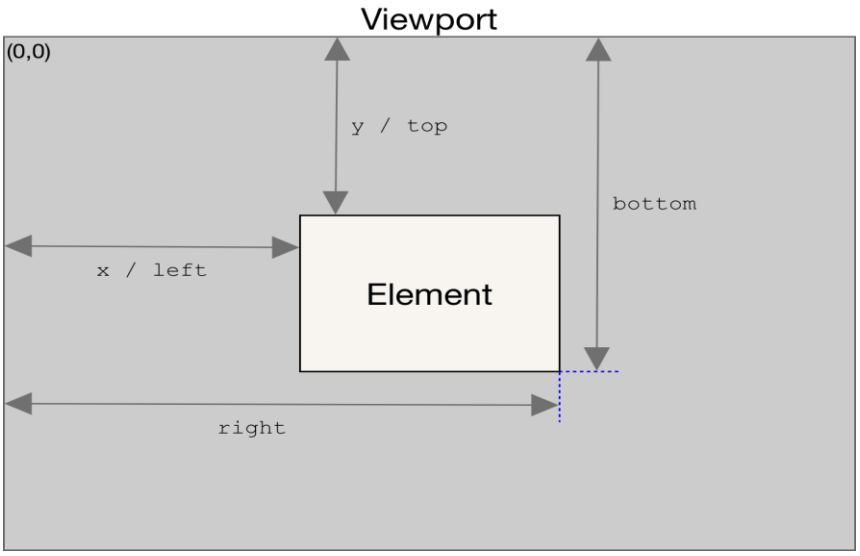
15.2.17 Element.value

The value property of the HTMLDataElement interface returns a DOMString reflecting the value HTML attribute.

15.2.18 Element.getBoundingClientRect()

The Element.getBoundingClientRect() method returns a DOMRect object providing information about the size of an element and its position relative to the viewport.

The returned value is a DOMRect object which is the smallest rectangle which contains the entire element, including its padding and border-width. The left, top, right, bottom, x, y, width, and height properties describe the position and size of the overall rectangle in pixels. Properties other than width and height are relative to the top-left of the viewport.



15.3 DOM Events

The user interacts with the HTML elements and each of these interactions is what we call as an event. Each of these events have predefined event attributes which link the JS code to these events. Some of the important ones are:

Event	Event Attribute	Description
click	onclick	The event occurs when the user clicks an HTML element.
load	onload	The event occurs when the browser has finished loading the page.
mouseover	onmouseover	The event occurs when the user moves the mouse over an HTML element
focus	onfocus	The event occurs when an element gets focus
blur	onblur	The event occurs when an element loses focus
keyup	onkeyup	The event occurs when the user releases a key

The JavaScript code(called specifically as function) is written to respond to these events are called as event handlers.

15.3.1 EventTarget.addEventListener()

The EventTarget method addEventListener() sets up a function that will be called whenever the specified event is delivered to the target.
Common targets are Element, Document, and Window, but the target may be any object that supports events (such as XMLHttpRequest).
addEventListener() works by adding a function or an object that implements EventListener to the list of event listeners for the specified event type on the EventTarget on which it's called.

```
target.addEventListener(type, listener, options);

target.addEventListener(type, listener, useCapture);
```

15.3.2 EventTarget.removeEventListener()

The EventTarget.removeEventListener() method removes from the EventTarget an event listener previously registered with EventTarget.addEventListener(). The event listener to be removed is identified using a combination of the event type, the event listener function itself, and various optional options that may affect the matching process;

15.3.3 Event Object

Events in JavaScript are considered as objects.
When Events are fired, the 'event' object is generated by the browser. This object encapsulates all data related to that event.
To access or manipulate this object, we can optionally pass it as the first argument to the event handler function.

```
<input type="radio" onclick="display(event)" value="male">Male
function display(e){
console.log(e);
}
```

Event Target

We can get details of the element on which the event took place using event.target

The target object gives the DOM object and from it we can get the various properties. For example, in the below code, we are accessing the name and value of the radio button which gets clicked:

```
<input type="radio" name="gender" value="male" onclick="display(event)">Male
<script>
   function display(e){
      console.log(e.target.name);
      console.log(e.target.value);
   }
</script>
```

Event Current Target

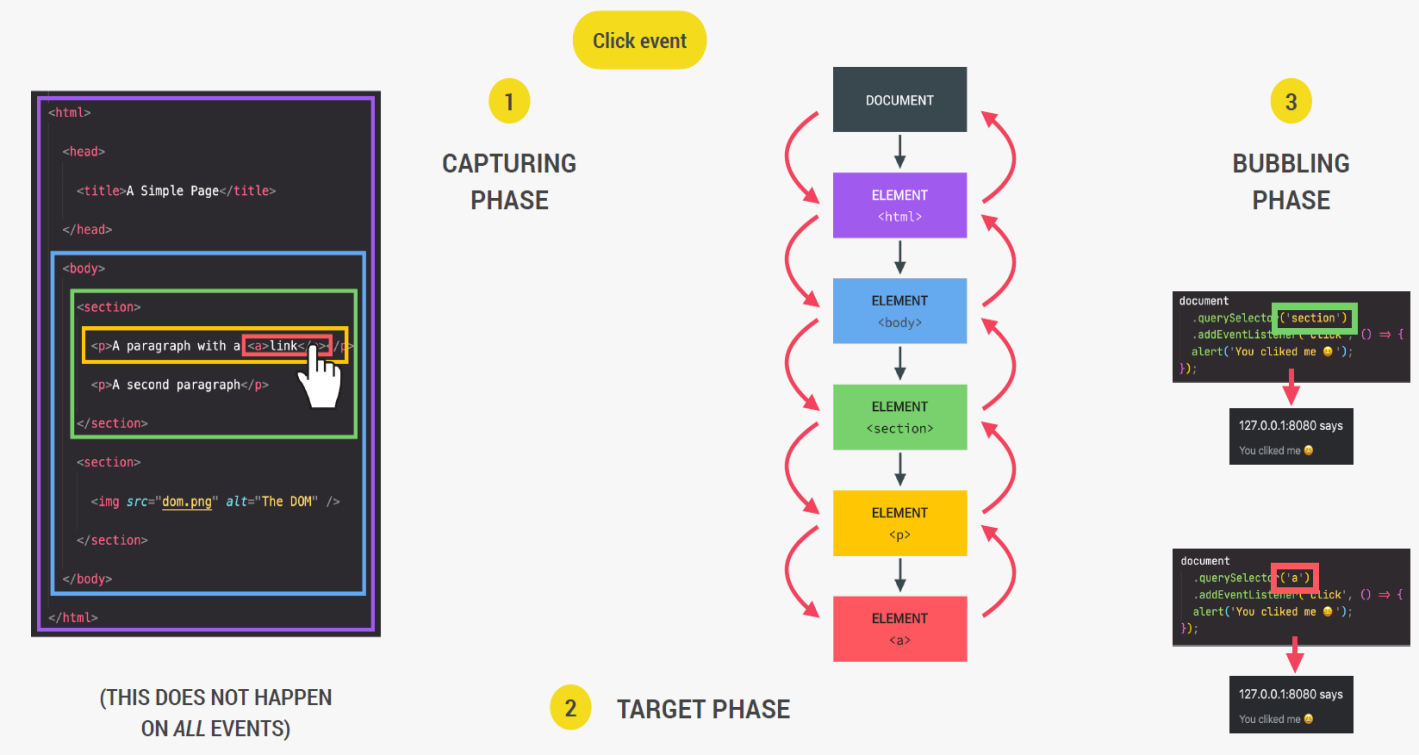
The currentTarget read-only property of the Event interface identifies the current target for the event, as the event traverses the DOM. It always refers to the element to which the event handler has been attached, as opposed to Event.target, which identifies the element on which the event occurred and which may be its descendant.

15.3.4 Event Capturing

Event capturing is the event taht starts from top element to the target element

15.3.5 Event Bubbling

When an event occurs on an element, that event 'bubbles' up to all its parent tags as well. This will in turn trigger the event handlers of those parent tags as well.




```
<div style="border:1px solid;padding:10px" onclick="div1Click(event)">
   Div 1
   <div style="border:1px solid;padding: 10px" onclick="div2Click(event)">
      Div 2
   </div>
</div>
<script>
   function div1Click(e){
      console.log("Div1 was clicked")
   }
   function div2Click(e){
      console.log("Div2 was clicked")
   }
</script>
```

Div2 is a child of Div1. Both of them have onclick event handlers which execute different functions. When Div2 is clicked, it executes the div2Click() and then the event bubbles up to its parent which is Div1.

Now the click event comes to Div1 and it executes div1Click() event handler as well. This is called as event bubbling.

The **stopPropagation()** method of the Event interface prevents further propagation of the current event in the capturing and bubbling phases.

```
function div2Click(e){
   e.stopPropagation();// stop event bubbling
   console.log("Div2 was clicked")
}
```

15.3.6 Event Prevention

Sometimes, when an event occurs, we may want to stop its further behaviour. For example, we may want to stop a page from getting submitted, if the validations fail or we may want to prevent event bubbling from happening. In such cases if we use event.preventDefault() it will prevent the default behaviour of the event.

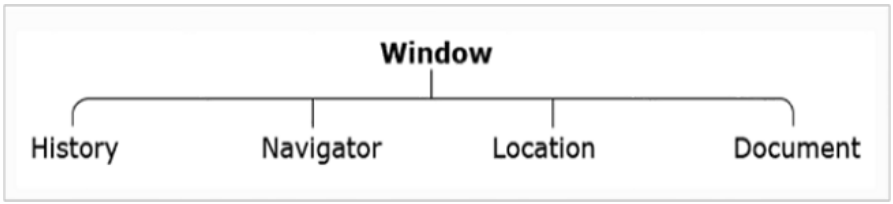
```
<form onsubmit="display(event)">
      Username: <input id="usr">
      <button type="submit">Login</button>
      <div id="msg"></div>
</form>
<script>
   function display(e){
      if(document.getElementById("usr").value.length<6){
         document.getElementById("msg").innerText="Username must be atleast 6 chars"
         e.preventDefault();
      }
   }
</script>
```

15.3.7 Event Delegation

When parent element automatically adds event to its child elements.
Event listener will fire anytime an event is triggered on child element(bubbling).

16. BOM

We have read earlier in the course that JavaScript is capable of dynamically manipulating the content and style of HTML elements of the web page currently rendered on the browser. The content given for para during HTML creation or the style given for heading during HTML creation can be changed even after the page has arrived on the browser.
This dynamic manipulation of an HTML page on the client-side itself is achieved with the help of built-in browser objects. They allow JavaScript code to programmatically control the browser and are collectively known as Browser Object Model (BOM).
For programming purposes, the BOM model virtually splits the browser into different parts and refers to each part as a different type of built-in object. BOM is a hierarchy of multiple objects. 'window' object is the root object and consists of other objects in a hierarchy, namely, 'history' object, 'navigator' object, 'location' object, and 'document' object.



16.1 History Object

If required, BOM also gives us a specific object to target only one of the window properties. For example, if we are only concerned about the list of URLs that have been visited by the user and we do not need any other information about the browser, BOM gives us the 'history' object for this. It provides programmatic navigation to one of the URLs previously visited by the user. The following are the properties or methods that help us do it.

Property:

length

returns the number of elements in the History list.
Usage: history.length;

Methods:

back()
method, loads previous URL from history list.
Usage: history.back();

forward()
method, loads next URL from history list. Usage: history.forward();

go()
method, loads previous URL present at the given number from the history list.

16.2 Navigation Object

It contains information about the client, that is, the browser on which the web page is rendered. The following properties and methods help us get this information.

appName

Returns the name of the client.

```
navigator.appName;
//Browser's name: Netscape
```

appVersion

Returns platform (operating system) and version of the client (browser).

```
console.log(navigator.appVersion);
//5.0 (Windows NT 10.0; Win64; x64)
//AppleWebKit/537.36 (KHTML, like Gecko)
//Chrome/83.0.4103.106 Safari/537.36
```

Platform

Returns the name of the user's operating system.

```
console.log(navigator.platform);
//Browser's platform: Win 32
```

userAgent

Returns string equivalent to HTTP user-agent request header.

```
console.log(navigator.userAgent);
//Browser's useragent: Mozilla/5.0 5.0 (Windows NT 6.1; WOW64)
//AppleWebKit/537.36 (KHTML, like Gecko)
//Chrome/53.0.2785.116 Safari/537.36
```

Geolocation

The Geolocation.getCurrentPosition() method is used to get the current position of the device.

```
navigator.geolocation.getCurrentPosition(position => {
  console.log(position)
})
```

```
navigator.geolocation.getCurrentPosition(success, error, [options])
```

success

A callback function that takes a GeolocationPosition object as its sole input parameter.

error Optional

An optional callback function that takes a GeolocationPositionError object as its sole input parameter.

options Optional

An optional PositionOptions object. Options includes:

maximumAge: integer (milliseconds) | infinity - maximum cached position age.

timeout: integer (milliseconds) - amount of time before the error callback is invoked, if 0 it will never invoke.

enableHighAccuracy: false | true

16.3 Location Object

if we want to programmatically refresh the current page or navigate to a new page which objects shall we use?
BOM hierarchy has a 'location' object for this. It contains information about the current URL in the browser window. The information can be accessed or manipulated using the following properties and methods.

If this is the URL: http://localhost:8080/JS_Demos/myLocationFile.html, properties have the following interpretation:

href

It contains the entire URL as a string.

```
console.log(location.href);
//Returns http://localhost:8080/JS_Demos/myLocationFile.html
```

hostname

It contains the hostname part of the URL.

```
console.log(location.hostname);
//Returns localhost
```

port

It contains a port number associated with the URL.

```
console.log(location.port)
```

```
//Returns 8080
```

pathname

It contains a filename or path specified by the object.

```
console.log(location.pathname);
//Returns /JS_Demos/myLocationFile.html
```

'location' object gives the following methods to reload the current page or to navigate to a new page:

assign()

Loads new HTML document.

```
location.assign('http://www.facebook.com');
//Opens facebook page
```

reload()

Reloads current HTML.

```
location.reload();
//Current document is reloaded
```

16.4 Window Object

Now, consider a scenario where you do not want to update the HTML page but only certain properties of the browser window on which it is rendered. Maybe, you want to navigate to a different URL and bring a new web page, or you want to close the web page or you want to store some data related to the web page. Well,

to implement this, we would need an object that represents the entire browser window and allows us to access and manipulate the window properties. BOM model provides us 'window' object.

This object resides on top of the BOM hierarchy. Its methods give us access to the toolbars, status bars, menus, and even the HTML web page currently displayed.

Window Object - Propeties

innerHeight

This property holds the inner height of the window’s content area.

innerWidth

This property holds the inner width of the window’s content area.

outerHeight

This property holds the outer height of the window including toolbars and scrollbars.

outerWidth

This property holds the outer width of the window including toolbars and scrollbars.

Window Object - Methods

localStorage

This property allows access to object that stores data without any expiration date.

```
localStorage.setItem('username','Bob');
console.log("Item stored in localStorage is" + localStorage.getItem('username'));
//Returns Item stored in localStorage is Bob
```

sessionStorage

This property allows access to objects that store data valid only for the current session.

```
sessionStorage.setItem('password', 'Bob@123');
console.log("Item stored in sessionStorage is " + sessionStorage.getItem('password'));
//Returns Item stored in sessionStorage is Bob@123
```

In addition to these methods, 'window' object gives us a few more methods that are helpful in the following way:
open() method, opens a new window. Usage: window.open("http://www.xyz.com");
close() method, closes the current window. Usage: window.close()

17. Classes

17.1 Constructor Pattern

You can also create Objects using Constructor Pattern.To use this approach developer need to use new keyword. It allows user to create custom Object through JavaScript built-in Object class as well as customized implementation.

For example let us create Doctor object using new keyword with the help of JavaScript built-in Object class.

```
// Doctor object creation using Object class.
doctor1 = new Object();
//To add dynamically properties
doctor1.name="Tom";
doctor1.specialization="E.N.T";
//code to access the declared property
console.log(doctor1.name);
console.log(doctor1.specialization);
```

Now let us create a customize Doctor object to implement same concept.

```
//customized Doctor object declaration through function
function Doctor(name,specialization)
{
this.name=name;
this.specialization=specialization;
this.availability=function() { return this.name +" " + "From Morning 9 A.M - 11 A.M"}
}
// Doctor object creation.
doctor1 = new Doctor("Tom","ENT");
//code to access the declared property
console.log(doctor1.name);
console.log(doctor1.specialization);
//code to invoke method through user created object
console.log(doctor1.availability());
```

Using ECMA Script 5 user can create customized object only through function, as JavaScript doesn't have class concept but from ECMA Script 6 and above this implementation can be directly achieved through class keyword.

17.2 Class

OOP Feature	Description
Class	Can be created using the **class** keyword
Constructor	Can be created using the **constructor** keyword
Attributes	Variables created as **this.<variableName>** inside the constructor become attributes
Methods	Functions created inside the class become methods
Object	Created using the **new** keyword
Access	The attributes and methods can be accessed using the **dot operator** on the object

The below code illustrates how to create a class with attributes and methods and how to create objects for the class.
this keyword is used to create attributes inside a class. This is equivalent to self in Python.

```
class Employee{
   constructor(id,name,age){
      this.id=id;
      this.name=name;
      this.age=age;
   }
   swipeIn(){
      console.log("Employee "+this.id+" has swiped in at "+new Date());
   }
}
e1=new Employee(100,"Mark",23);
e2=new Employee(101,"Jane",24);
console.log(e1.age);
e1.swipeIn();
e2.swipeIn();
```

Static methods in class

Just like in other programming languages, we can create static methods in JavaScript using the static keyword. Static values can be accessed only using the classname and not using this keyword. Else it will lead to an error.

In the below example, code is a static method and it is accessed using the classname.

```
class Employee{
   constructor(id,name,age){
      this.id=id;
      this.name=name;
      this.age=age;
   }
   swipeIn(){
      console.log("Employee "+this.id+" has swiped in at "+new Date());
   }
   static code(){
      console.log("Employee is coding");
   }
}
Employee.code();
```

17.3 Prototype and Prototype Chaining

In JavaScript all the Object are always linked to another JavaScript Object. This linked Object is called as prototype Object. Thus every JavaScript Objects inherits the properties from its associated prototype Object.

Functions are also Objects and they also have prototypes. We can find the prototype of a function by using the prototype property.

All Objects created through a Constructor Function also inherits from the prototype object of respective associated constructor (super/parent) function.

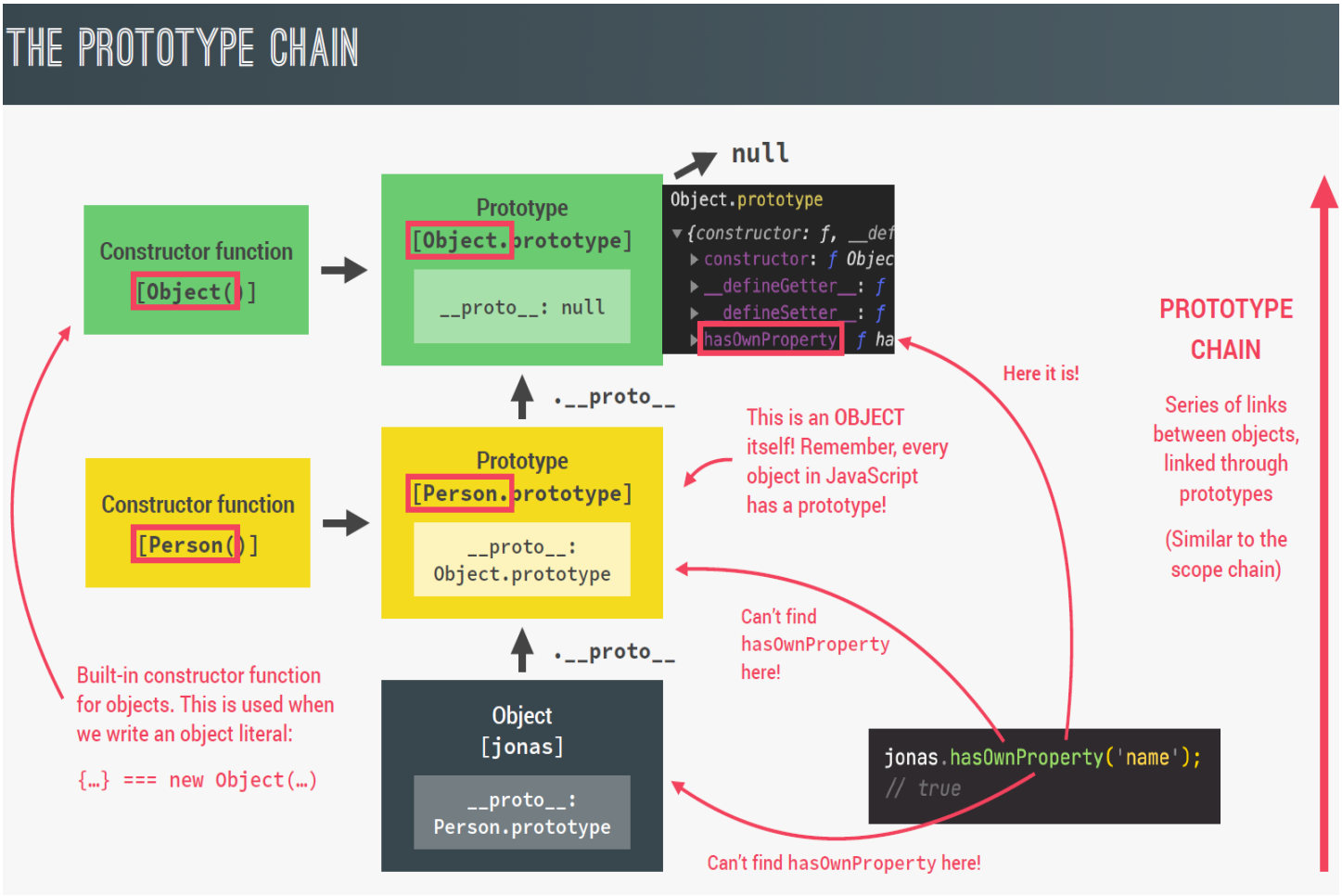
For example, in the below code we are accessing the prototype property of the Doctor constructor function as well as for adding dynamically hospital attribute to same constructor.

```
function Doctor(name, specialization) {
   this.name = name,
      this.specialization = specialization,
      this.availability = function() {
         return this.name + "From Morning 9 A.M - 11 A.M"
      }
}
// Doctor object creation.
doctor1 = new Doctor("Tom", "ENT");
//code to access the prototype property
console.log(Doctor.prototype);
//code to add hospital attribute dynamically through prototype object
Doctor.prototype.hospitalLocation = "Chennai";
console.log(Doctor.prototype.hospitalLocation);
```

Please note that dynamically added property like hospitalLocation would hold static value, which can be accessed by all the instances of the Doctor class.

As seen above by using Prototype Object developer can add new attributes , delete attributes and even define new method without modifying the declared structure.

```
function Doctor()
{}
// code to add new attributes and method
Doctor.prototype.name="Tom";
Doctor.prototype.specialization="ENT";
Doctor.prototype.availability = function() { return this.name +" " + "From Morning 9 A.M - 11 A.M"}
// Doctor object creation.
doctor1 = new Doctor();
//code to invoke availability method
console.log(doctor1.availability());
// code to delete the attribute
console.log(delete doctor1.name);
```



__proto__ is an object in every class instance that points to the prototype it was created from.

17.4 Object.create()

The Object.create() method creates a new object, using an existing object as the prototype of the newly created object.

```
Object.create(proto, propertiesObject)
```

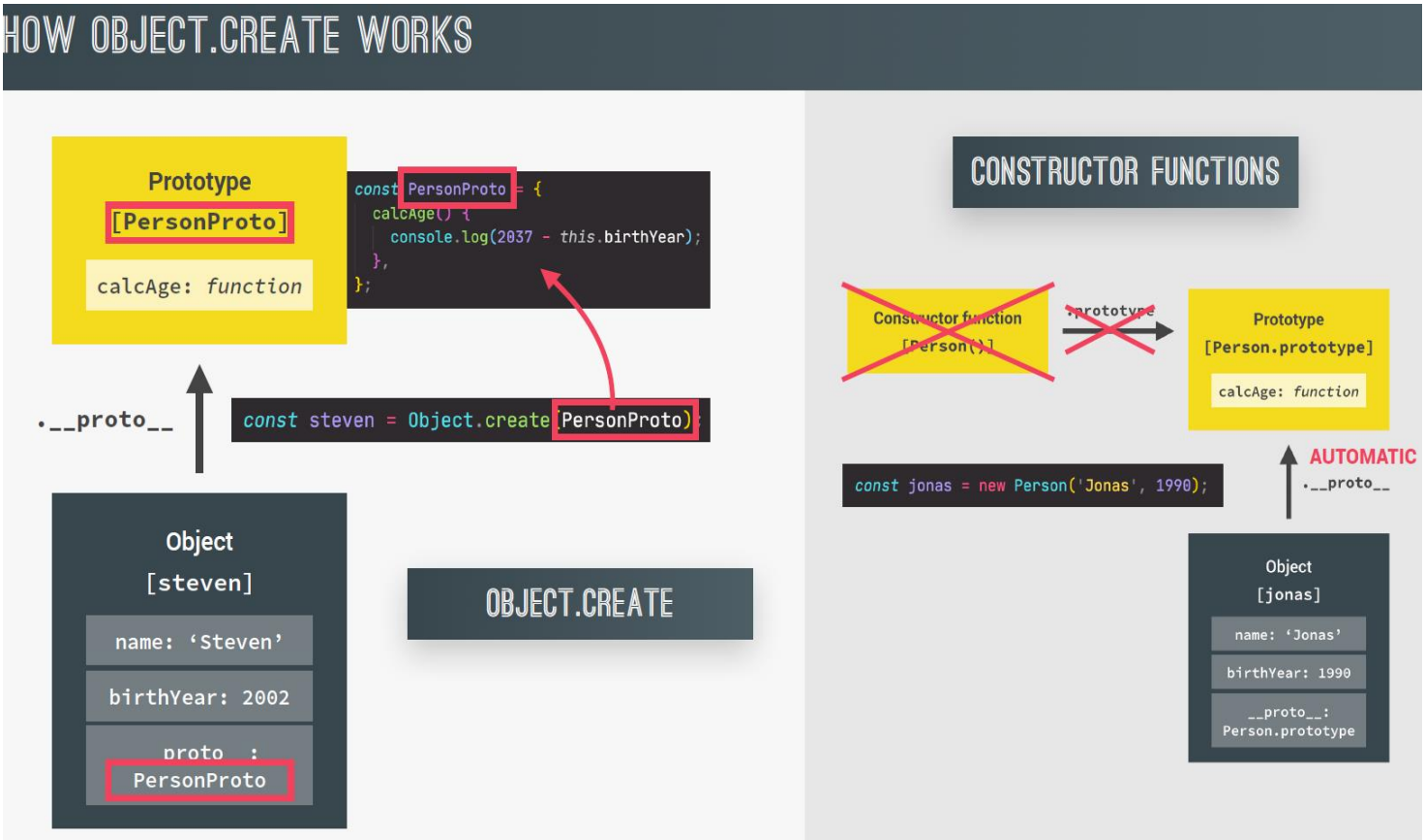
Parameters

proto

The object which should be the prototype of the newly-created object.

propertiesObject Optional

If specified and not undefined, an object whose enumerable own properties (that is, those properties defined upon itself and not enumerable properties along its prototype chain) specify property descriptors to be added to the newly-created object, with the corresponding property names.



17.5 Inheritance

17.5.1 Prototype Inheritance

Developer can inherit from one constructor to another constructor through Prototype Pattern.

```
function Doctor(name, specialization) {
   this.name = name,
      this.specialization = specialization,
      this.availability = function() {
         return this.name + " " + "From Morning 9 A.M - 11 A.M"
      }
}
function SpecializedDoctor(hospitalLocation) {
     //code to invoke parent class
    Doctor.call(this, name, specialization);
   this.hospitalLocation = hospitalLocation;
}
//code for establishing inheritance between SpecializedDoctor and Doctor
SpecializedDoctor.prototype = Object.create(Doctor.prototype);
SpecializedDoctor.prototype.constructor= SpecializedDoctor;

//code to create child object and invoking its parent function
sDoctor1 = new SpecializedDoctor("ABC Hospital");
console.log(sDoctor1.hospitalLocation);
console.log(sDoctor1);
```

17.5.2 Inheritance between classes

In JavaScript, one class can inherit another class using the extends keyword. The subclass inherits all the methods (both static and non-static) of the parent class.
Inheritance enables the reusability and extensibility of a given class.
JavaScript uses prototypal inheritance which is quite complex and unreadable. But, now we have the more friendly extends keyword which makes it easy to inherit the existing classes.
Keyword super can be used to refer to base class methods/constructors from a subclass

```
class Vehicle {
	constructor(make, model) {
		/* Base class Vehicle with constructor initializing two-member attributes */
		this.make = make;
		this.model = model;
	}
}
class Car extends Vehicle {
	constructor(make, model, regNo, fuelType) {
		super(make, model); // Sub class calling Base class Constructor
		this.regNo = regNo;
		this.fuelType = fuelType;
	}
	getDetails() {
		/* Template literals used for displaying details of Car. */
		console.log(`${this.make},${this.model},${this.regNo},${this.fuelType}`);
	}
}
let c = new Car("Hundai", "i10", "KA-016447", "Petrol"); // Creating a Car object
c.getDetails()
```

Note that super keyword must appear before this keyword in constructor.

17.6 Encapsulation

Encapsulation is the packing of data and functions into one component (for example, a class) and then controlling access to that component to make a "blackbox" out of the object. Because of this, a user of that class only needs to know its interface (that is, the data and functions exposed outside the class), not the hidden implementation.

Encapsulation in JavaScript can be achieved through var keyword.

By using var with any attribute declared in function, scope of that variable/attribute would be private to that declared block. To access these private scope attributes developer needs to write accessor (getters) and mutator (setters) methods.

Now let us modify previous defined Doctor function, and make name attribute as private and create getter and setter for the same attribute.

```
function Doctor(name) {
    //private attribute declaration
    var doctorName = name;
    //getter method declaration
    this.getName = function() {
        return doctorName;
    }
    //setter method declaration
    this.setName = function(name) {
        this.doctorName = name;
    }
    this.availability = function() {
        return this.doctorName + " " + "From Morning 9 A.M - 11 A.M"
    }
}
// Doctor object creation and invoking getter and setter methods.
doctor1 = new Doctor("Tom");
console.log(doctor1.getName());
doctor1.setName("Jim");
console.log(doctor1.getName());
console.log(doctor1.availability());
```

18. Asynchronous Programming

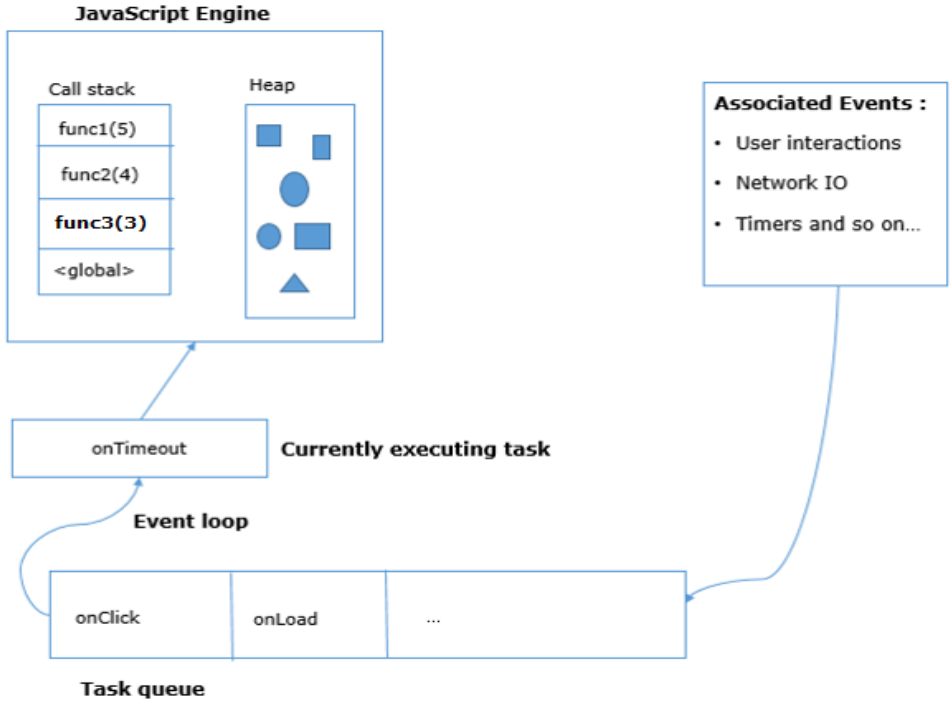
18.1 Single Thread Event Loop Architecture

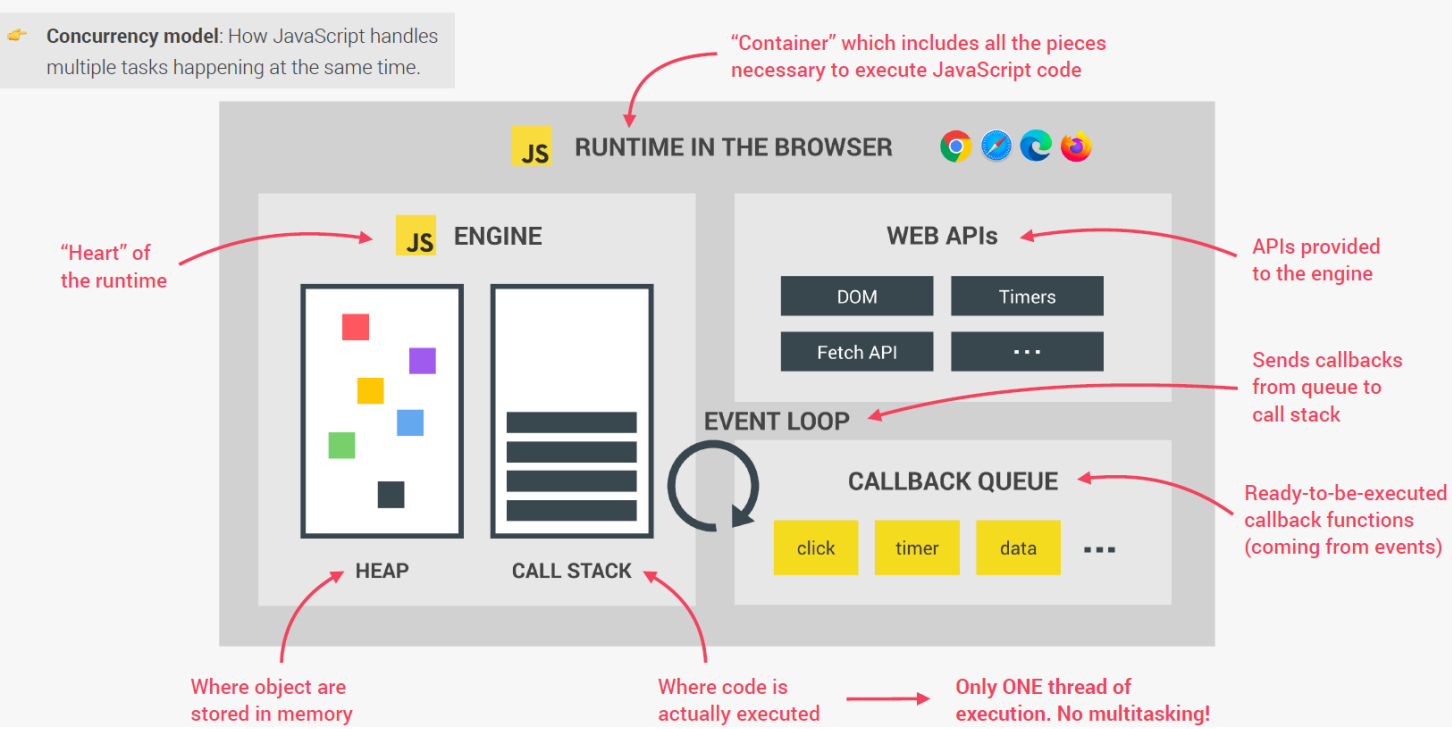
JavaScript environment :

- Executes only one thread at a time, internally it executes a single piece of code associated with that current executing method or function or chunk.
- By default concurrency is supported, due to this feature race condition is automatically handled by the system.
    - If there is any shared resource among two event handlers, automatically system executes only one event handler at run time.
- Internally JavaScript engine will
    - Maintains a queue – task queue, in which all the event which are associated with event handler business logic are placed.
    - Pulls the events from queue, based on user action.
    - Retrieves the event handler logic
    - Invoke them sequentially based on their queue position
    - Execute the respective associated logic and process the immediate event of the queue.

Please note that JavaScript container can pipeline events to the existing queue like mouse click, Network IO process, Timers, Interrupts and so on..

Now let us have a glance over Scenario 2 in pictorial representation:



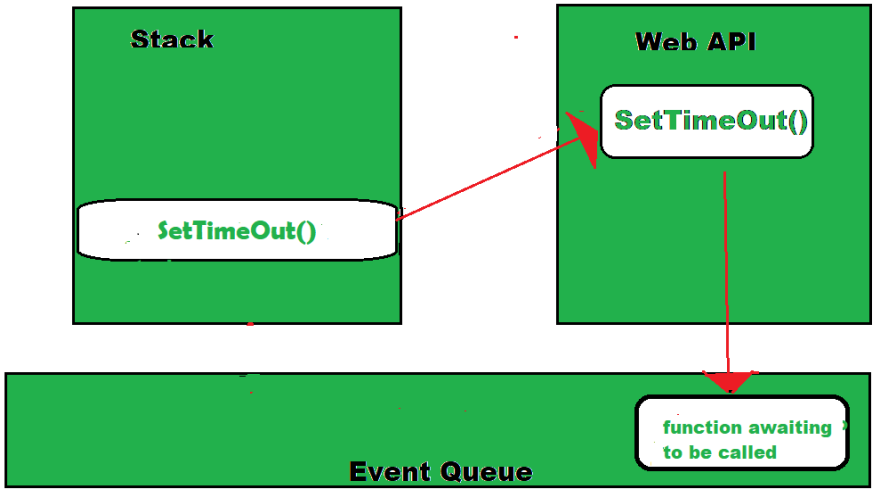


.

Event loop:

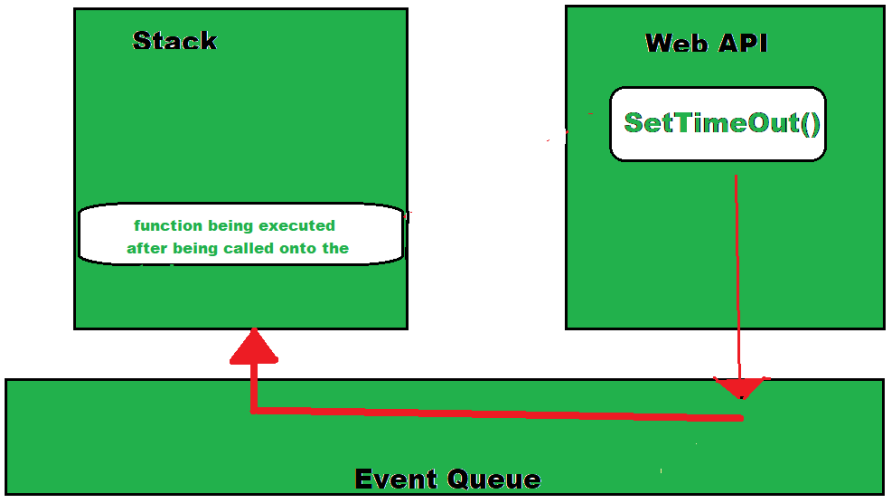
An event loop is something that pulls stuff out of the queue and places it onto call stack whenever the stack becomes empty.

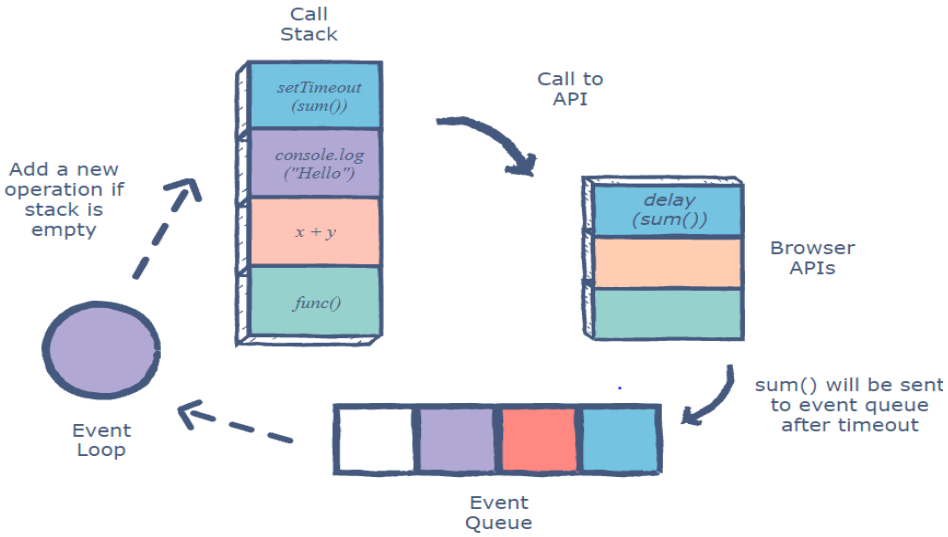
The event loop is the secret by which JavaScript gives us an illusion of being multithreaded even though it is single-threaded. The below illusion demonstrates the functioning of event loop well:



Here the callback function in the event queue has not yet run and is waiting for its time into the stack when the SetTimeOut() is being executed and the Web API is making the mentioned wait. When the function stack becomes empty, the function gets loaded onto the stack .

That is where the event loop comes into picture, it takes the first event from the Event Queue and places it onto the stack i.e in this case the callback function. From here, this function executes calling other functions inside it, if any.





18.2 Asynchronous Execution Implementation

setTimeOut()

- Is a method of the WindowOrWorkerGlobalScope mixin (which is successor to window.setTimeout)
- Helps in associating a timer with a single piece of code, which would be automatically executed once timer interval/delay expires.
- Internally this method adds a timeout event to the Task Queue with the associated event handler.

```
console.log("Before For loop execution");
for (var i = 0; i < 2; i++) {
    console.log("setTimeout message");
    func1();
    func2();
}
console.log("After For loop execution");
function func1() {
    console.log("Am in func1");
}
function func2() {
    console.log("Am in func2");
}
```

According to JavaScript sequentially execution nature output populates as shown below:

```
Before For loop execution
setTimeout message
Am in func1
Am in func2
setTimeout message
Am in func1
Am in func2
After For loop execution
```

Modified code snippet:

```
for (var i = 0; i < 2; i++) {
    setTimeout(function() {
        console.log("setTimeout message");
        func1();
    }, );
    func2();
}
```

```
Before For loop execution
Am in func2
After For loop execution
setTimeout message
Am in func1
setTimeout message
Am in func1
```

When you have glance over above shown output, due to usage of setTimeout() method the entire execution of code behavior has been changed, internally parallel/multi-threading concept has been invoked.

Let us achieve asynchronous execution through setInterval() as shown below:

```
console.log("Before For loop execution");
var counter = 0;
var t;
t = setInterval(function() {
    console.log("setTimeout message");
    func1();
    counter += 1;
    if (counter >= 3) {
        clearInterval(t);
    }
}, 1000);
func2();
console.log("After For loop execution");
function func1() {
    console.log("Am in func1");
}
function func2() {
    console.log("Am in func2");
}
```

```
Before For loop execution
Am in func2
After For loop execution
setTimeout message
Am in func1
setTimeout message
Am in func1
setTimeout message
Am in func1
```

By observing above output, you could find that setInterval() works similar to setTimeout() method, but the difference is setInterval() is business logic gets executed multiple times and to kill the timer you need to invoke clearTimer().
Please note that both of these methods are the only native functions to implement asynchronous execution in JavaScript environment.

Some of the real-time situation where you may use these methods are:

- To make any HTTP request call.
- To perform any Input/Output operations.
- When you deal with Client and Server communication.

Let us learn about Asynchronous pattern and its implementation in detail.

Please note that setInterval() works similar to setTimeout() method, but the difference is setInterval() is business logic gets executed multiple times and to kill the timer you need to invoke clearTimer().

These executions in JavaScript can also be achieved through many techniques.

- Callbacks
- Promises and
- Async/Await
- Fetch
- AJAX

18.3 Callback

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

Let us use callback to address the problem of dealing with data in an asynchronous situation. Consider the below code:

```
let value;
function check(val) {
   console.log(val);
}
function getTrip() {
   setTimeout(function () {
      value = "Let's go to Trip";
   }, 1500);
}
getTrip();
check(value);// undefined
```

This can be modified using callback as shown below:

```
let value;
function check(val) {
   console.log(val);//"Let's go to Trip";
}
function getTrip(callback) {
   setTimeout(function () {
      value = "Let's go to Trip";
      callback(value);
   }, 1500);
}
getTrip(check);
```

Here we are passing check function as a callback to the getTrip() function. Now check() will be invoked inside setTimeout() instead of returning the data.

Typically third party libraries expect you to pass a callback and they will invoke your callback after they have completed the task.

What if a callback has a callback? While doing multiple asynchronous operations, the callback get nested to each other which leads to callback hell.

Callback function can not be chained together which leads to nested callback while calling it multiple time which leads to callback hell.

```
myFun1(function () {
   myFun2(function () {
      myFun3(function () {
         myFun4(function () {
            ....
         });
      });
   });
});
```

In the above example, we can notice that the "pyramid" of nested calls grows to the right with every asynchronous action. It leads to callback hell. So, this way of coding is not very good practice.

To overcome the disadvantage of callbacks, the concept of Promises was introduced.

18.4 Promise

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

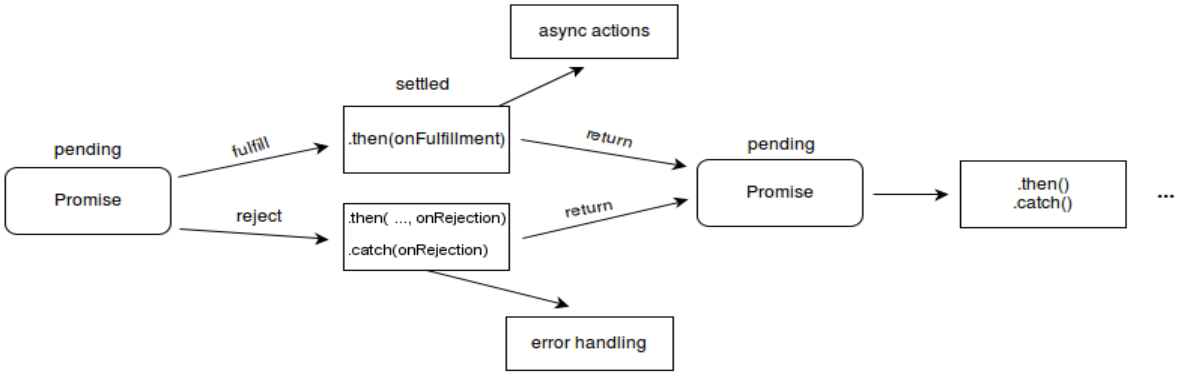
```
new Promise( /* executor */ function(resolve, reject) { ... } );
```

executor:

- Is a function, which accepts two arguments namely resolve and reject.
- It gets executed before the Promise constructor invoked the code and would return object.
- Internally this object would either hold success (resolve) or error (reject) value.
- Resolve and Reject arguments basically represent asynchronous business logic code, which should be automatically triggered on success or failure value respectively.

A Promise is in one of these states:

- pending: initial state, neither fulfilled nor rejected.
- fulfilled: meaning that the operation was completed successfully.
- rejected: meaning that the operation failed.



18.4.1 Promise Methods

Methods	Usage
Promise.prototype.catch(onRejected), Promise.prototype.finally(onFinally)	• These methods can be used for exception handling concepts.
Promise.prototype.then(onFulfilled, onRejected)	• Helps you in handling fulfill and reject mechanism based on Promise object state.

```
var myPromise = new Promise(function (resolve, reject) {
	setTimeout(function () {
		resolve("success");
	}, 2000);
});

myPromise.then(
	function (data) {
		console.log(data + " received in 2 seconds");
	},
	function (error) {
		console.log(error);
	}
);

	OR

myPromise.then(
	function (data) {
		console.log(data + " received in 2 seconds");
	})
.catch(
	function (error) {
		console.log(error);
	}
);
```

18.4.2 Promise Chaining

The 'Callback hell', is now resolved using 'Chaining' which creates readable code and is an eminent feature of Promise. Here, the asynchronous code can be chained using multiple then statements.

If we fulfil the first promise, second promise will make sense and vice-versa for reject.

```
doSomething().then(function (result) {
        return doSomethingElse(result);
})
        .then(function (newResult) {
                return doThirdThing(newResult);
        })
        .then(function (finalResult) {
                console.log("Print the final result " + finalResult)
        })
        .catch(failureCallBack);
```

18.4.3 Promise Property

Methods	Usage
Promise.all(iterable)	• This method helps in returning Array of Promise objects. • Returns either fullfills or reject value. • If value is fulfills – when all the promises in iterable argument are in fulfills state and • If value is reject – when first promise object of iterable argument is in reject state and returns error message.
Promise.race(iterable)	• This method helps in implementing Timing concept. • Returns Promise by accepting an iterable over Promises through thenable's and other values which would be converted to Promises through Promise.resolve(). • Never settled when the first Promise object of iterable argument is empty
Promise.reject(reason)	• Returns the error reason of rejected Promise object.

The Promise.all() method is actually a promise that takes an array of promises(an iterable) as an input. It returns a single Promise that resolves when all of the promises passed as an iterable, which have resolved or when the iterable contains no promises. In simple way, if any of the passed-in promises reject, the Promise.all() method asynchronously rejects the value of the promise that already rejected, whether or not the other promises have resolved.

The Promise.all() static method accepts a list of Promises and returns a Promise that:

- resolves when every input Promise has resolved or
- rejected when any of the input Promise has rejected.

```
Promise.all( iterable )
```

The Promise.allSettled() method returns a promise that resolves after all of the given promises have either fulfilled or rejected, with an array of objects that each describes the outcome of each promise.

It is typically used when you have multiple asynchronous tasks that are not dependent on one another to complete successfully, or you'd always like to know the result of each promise.

```
Promise.allSettled(iterable);
```

```
// Illustration of Promise.allSettled()
// Method in Javascript with Example

const p1 = Promise.resolve(50);
const p2 = new Promise((resolve, reject) =>
            setTimeout(reject, 100, 'geek'));
const prm = [p1, p2];

Promise.allSettled(prm).
  then((results) => results.forEach((result) =>
  console.log(result.status,result.value)));
```

Output:

```
"fulfilled"
 50
"rejected"
 Undefined
```

The Promise.race() method returns a promise that fulfills or rejects as soon as one of the promises in an iterable fulfills or rejects, with the value or reason from that promise.

```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, 'one');
});

const promise2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'two');
});

Promise.race([promise1, promise2]).then((value) => {
  console.log(value);
  // Both resolve, but promise2 is faster
});
// expected output: "two"
```

The Promise.any() method accepts a list of Promise objects as an iterable object. As soon as a Promise from the list fulfills, the Promise.any() returns the fulfilled Promise that resolves with a value.
If no promises in the iterable fulfill, the Promise.any() rejects with an AggregateError that groups individual errors. The AggregateError is a subclass of Error.
Essentially, the Promise.any() method is the opposite of the Promise.all() method.

```
const promise1 = Promise.reject(0);
const promise2 = new Promise((resolve) => setTimeout(resolve, 100, 'quick'));
const promise3 = new Promise((resolve) => setTimeout(resolve, 500, 'slow'));
const promises = [promise1, promise2, promise3];
Promise.any(promises).then((value) => console.log(value));
// expected output: "quick"
```

18.5 Async/Await

"async/await" was introduced to implement asynchronous code with promises that resemble synchronous code. We can say that async/await is simple, easy, and more readable and understandable than the promises.

Async/Await vs Promises

	Async/Await	**Promises**
Scope	The entire wrapper function is asynchronous	Only the promise chain itself is asynchronous
Logic	• Synchronous work needs to be moved out of the callback • Multiple promises can be handled with simple variables	• Synchronous work can be handled in the same callback • Multiple promises use Promise.all()
Error Handling	We can use Try, Catch and Finally	We can use Then, Catch, Finally

18.5.1 Async Function

An async function is declared with an async keyword. It always returns a promise and if the value returned is not a promise, the JavaScript will automatically wrap the value in a resolved promise.

```
async function hello() {
   //Value will be wrapped in a resolved promise and returned
   return "Hello Async";
}

hello().then(val => console.log(val)); // Hello Async

async function hello() {
   //Promise can be returned explicitly as well
   return Promise.resolve("Hello Async");
}
hello().then(val => console.log(val)); // Hello Async
```

18.5.2 Await

Await keyword makes JavaScript wait until the promise returns a result. It works only inside async functions.

JavaScript throws Syntax error if await is used inside regular functions.

Await keyword pauses only the async function execution and resumes when the Promise is settled.

```
function sayAfter2Seconds(x) {
	return new Promise(resolve => {
		setTimeout(() => {
			resolve(x);
		}, 2000);
	});
}
async function hello() {
	//wait until the promise returns a value
	var x = await sayAfter2Seconds("Hello Async/Await");
	console.log(x); //Hello Async/Await
}
hello();
```

18.6 AJAX

Abbreviation of AJAX is Asynchronous JavaScript And XML.

It can be considered as a client-side scripting concept, which can be used to communicate with Servers through XMLHttpRequest Object.

For any interaction User can initiate the communication using different data formats like plain text, HTML, XML and JSON.

Due to its basic Asynchronous nature, any transaction involved through this concept wouldn't reload the entire page data but indeed it updates only need content based on user's request.

The JavaScript code can asynchronously connect to a server by using a new XMLHttpRequest() object. This is also called as AJAX.

The four steps for using AJAX are:

- Create new XMLHttpRequest()
- Open a URL using the request object
- Mention what should happen when a response is received
- Actually send the request

Properties	Usage
readyState	**There are 5 states for a request :** 0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready
onreadystatechange	holds a function (or the name of a function) to be called automatically each time the readyState property changes
status	status returns the current state/result of the request sent (e.g. "503" for "Service Unavailable" or "200" for "OK")
statusText	Returns the status-text (e.g. "Not Found" or "OK")
responseXML	Returns the response data as XML data
responseText	Returns the response data as a String

Methods	Usage
setRequestHeader()	It carries the information about expected response, who send the request and body of the request
open(method, url, async call)	It defines the complete prototype of the request and it hold three parameters like: • method: the type of request: GET or POST • url: the location of the file on the server • async call: true (asynchronous) or false (synchronous)
send()	Sends the request to the server
abort()	Withdraw the current request
setRequestHeader()	Gives the specific header information

```
var xhr = new XMLHttpRequest(); // 1.Create request object

xhr.open('GET', url); // 2.Open the URL
xhr.onload = function () { // 3.Mention code to run when response is received
  console.log("The response from server is "+xhr.responseText);
};
xhr.send(); // 4.Send the Request
```

```
// to establish the connection
XHR.open(method, url,async call);
// binding anonymous function
XHR.onreadystatechange = function()
{
// checking the state and status of XHR object, on success adding
content to respective DOM element
   if( XHR.readyState == 4 && XHR.status == 200 )
   {
   document.getElementById("<<element-name>").innerHTML= XHR.responseText;
   }
}
// to send back the request to the server
XHR.send()
```

18.7 Fetch

The Fetch API provides a fetch() method defined on the window object, which you can use to perform requests. This method returns a Promise that you can use to retrieve the response of the request.

The fetch method only has one mandatory argument, which is the URL of the resource you wish to fetch.

It returns a Promise that resolves to Response if the fetch operation is successful or throws an error that can be caught in case the fetch fails. You can also optionally pass in an init options object as the second argument.

The Promise returned from fetch() **won't reject on HTTP error status** even if the response is an HTTP 404 or 500. Instead, it will resolve normally (with ok status set to false), and it will only reject on network failure or if anything prevented the request from completing.

Syntax:

PromiseReturned = fetch(urlOfTheSite, [options])

- urlOfTheSite – The URL to be accessed.
- options – optional parameters: method, headers, etc.

Without options, this is a simple/default GET request which downloads the contents from the URL. The fetch() returns a promise which needs to be resolved to obtain the response from the server or for handling the error.

Getting a response from a fetch() is a two-step process.

1. The promise object returned by fetch() needs to be resolved to an object after the server sends a response.

Here, HTTP status needs to be checked to see it is successful or not.
The promise will be rejected if the fetch is unable to make a successful HTTP-request to the server e.g. may be due to network issues, or if the URL mentioned in fetch does not exist.
HTTP-status can be seen in response properties easily by doing console.log

- status – HTTP status code returned from a response, e.g. 200.
- ok – Boolean, true if the HTTP status code returned from a response, is 200-299.

2. Get the response body using additional methods.

Response object can be converted into various formats by using multiple methods to access the body/data from response object:

- response.text() –read body/data from response object as a text,
- response.json() – parse body/data from response object as JSON,
- response.formData() – return body/data from response object as FormData
- response.blob() – return body/data from response object as Blob (binary data with it's type)

//pass any url that you wish to access to fetch()

```
let response = await fetch(url);
if (response.ok) { // if HTTP-status is 200-299
   // get the response body
   let json = await response.json();
   console.log(json)
}
else {
   console.log("HTTP-Error: " + response.status);
}
```

To send a POST request, use the following code:

```
const params = {
   param1: value1,
   param2: value2;
};
const options = {
   method: 'POST',
   headers: {
      "Content-type": "application/json
   },

   body: JSON.stringify( params )
};
fetch( 'https://domain.com/path/', options )
   .then( response => response.json() )
   .then( response => {
      // Do something with response.
   } );
```

fetch() allows you to make network requests similar to XMLHttpRequest (XHR). The main difference is that the Fetch API uses Promises, which enables a simpler and cleaner API, avoiding callback hell and having to remember the complex API of XMLHttpRequest.

- Created on xhr with built-in promises.
- Compact syntax than xhr.

19. Modular Programming

Modules are one of the most important features of any programming language.
In 2015 modules were introduced in JavaScript officially and they are considered to be first-class citizens while coding the application.
Modules help in state and global namespace isolation and enable reusability and better maintainability.
We need modules in order to effectively reuse, maintain, separate, and encapsulate internal behavior from external behavior.
Each module is a JavaScript file.
Modules are always by default in strict-mode code. That is the scope of the members (functions, variables, etc.) which reside inside a module is always local.
The functions or variables defined in a module are not visible outside unless they are explicitly exported.
The developer can create a module and export only those values which are required to be accessed by other parts of the application.

Modules are declarative in nature:

- The keyword "export" is used to export any variable/method/object from a module.
- The keyword "import" is used to consume the exported variables in a different module.

19.1 Export

The export keyword is used to export some selected entities such as functions, objects, classes, or primitive values from the module so that they can be used by other modules using import statements.

There are two types of exports:

- Named Exports (More exports per module)
- Default Exports (One export per module)

19.1.1 Named Exports

Named exports are recognized by their names. We can include any number of named exports in a module.

We can export entities from a module in two ways.

Export individual features

```
export let name1, name2, …, nameN; // also var, const

export let name1 = …, name2 = …, …, nameN;

export function functionName(){...}

export class ClassName {...}
```

```
export let var1,var2;
export function myFunction() { ... };
```

Export List

```
export { name1, name2, …, nameN };
```

```
export { myFunction, var1, var2 };
```

19.1.2 Export Default

The most common and highly used entity is exported as default. We can use only one default export in a single file.

```
export default entityname;
```

where entities may be any of the JavaScript entities like classes, functions, variables, etc.

```
export default function () { ... }
export default class { .. }
```

We may have both default and named exports in a single module.

19.2 Import

If you want to utilize an exported member of a module, use the import keyword. We can use many numbers of import statements.

19.2.1 How to import Named Exports

```
//import multiple exports from module
import {entity1, entity 2... entity N} from modulename;
//import an entire module's contents
import * as variablename from modulename;
//import an export with more convenient alias
import {oldentityname as newentityname } from modulename;
```

```
import {var1,var2} from './mymodule.js';
import * as myModule from './mymodule.js';
import {myFunction as func} from './mymodule.js';
```

19.2.2 How to import Default Exports

We can import a default export with any name.

```
import variablename from modulename;

import myDefault from './mymodule.js';
```

Note:

- There can be only one default export per file
- While importing one should NOT use { } for default export. Else it will cause an error.

20. Browser Storage

We access all web content using HTTP.
HTTP describes a set of rules for smooth communication between client and server. However, HTTP is stateless. Stateless means, the server does not know which client is making the request. If the same client makes multiple requests, the server will not know that the requests are coming from the same client.
When you open Facebook or Gmail, you get only your information. That means somehow the server knows how to identify you and supply your data out of billions of users.
How does the server identify the client then, if HTTP is stateless?

Typically, the clients send some information to the server along with the request, which identifies them. This is done by using technologies like sessions, cookies, URL encoding, etc. But in all such cases, we have to pass information along with requests to the server, thus increasing the load on the server.

HTML5 introduced a concept called Web Storage API.
Web Storage is basically a local storage space in the clients hard disk. The data is stored in the clients' local storage rather that being sent to the server.
For example, if you add items to a cart on a shopping app, they can be stored in the local storage rather than being sent to the server.
Thus, when the page loads again, the data can be retrieved from the local storage itself, thereby reducing load on the server.

Note: Each website has a separate localstorage area allotted to them in the clients machine. Therefore Google cannot access what Amazon has stored in the clients localstorage. Also each browser has a different localstorage. That means what Google stores in localstorage of a client machine using chrome, will not be able to access that data if the user logins from Internet Explorer as IE has its own localstorage.

Featues of Web Storage:

- Large amount of data can be stored – Atleast 5MB data can be stored in client machine.
- Reduced network overhead – Stored data is never send back to web server. Hence, there are no additional HTTP request-response cycle.
- Secure – Each domain is given a part of memory in client's machine. Webpages from same domain can share data. One domain cannot override/access data of another domain. Hence, it is ensures data security.

20.1 Session Storage

For particular session i.e. session storage: For session storage, sessionStorage object is used.

Data is stored in the form of (key, value) pair and both key and value are stored as String.

Method	Description
setItem(key, value)	Stores value associated with key
getItem(key)	Retrieves value associated with the key
removeItem(key)	Removes value associated with the key
clear()	Clears all (key, value) pair

20.2 Local Storage

Across sessions i.e. local storage: For local storage, localStorage object is used.

Data is stored in the form of (key, value) pair and both key and value are stored as String.

Method	Description
setItem(key, value)	Stores value associated with key
getItem(key)	Retrieves value associated with the key
removeItem(key)	Removes value associated with the key
clear()	Clears all (key, value) pair

localStorage and sessionStorage are almost identical and have the same API. The difference is that with sessionStorage, the data is persisted only until the window or tab is closed. With localStorage, the data is persisted until the user manually clears the browser cache or until your web app clears the data.

20.3 Cookies

An HTTP cookie (web cookie, browser cookie) is a small piece of data that a server sends to a user's web browser. The browser may store the cookie and send it back to the same server with later requests. Typically, an HTTP cookie is used to tell if two requests come from the same browser—keeping a user logged in, for example. It remembers stateful information for the stateless HTTP protocol.

20.3.1 Creating Cookies

After receiving an HTTP request, a server can send one or more Set-Cookie headers with the response. The browser usually stores the cookie and sends it with requests made to the same server inside a Cookie HTTP header. You can specify an expiration date or time period after which the cookie shouldn't be sent. You can also set additional restrictions to a specific domain and path to limit where the cookie is sent. For details about the header attributes mentioned below, refer to the Set-Cookie reference article.

The Set-Cookie HTTP response header sends cookies from the server to the user agent. A simple cookie is set like this:

```
document.Cookie: <cookie-name>=<cookie-value>
```

Define the lifetime of a cookie

The lifetime of a cookie can be defined in two ways:

- Session cookies are deleted when the current session ends. The browser defines when the "current session" ends, and some browsers use session restoring when restarting. This can cause session cookies to last indefinitely.
- Permanent cookies are deleted at a date specified by the Expires attribute, or after a period of time specified by the Max-Age attribute.

```
document.Cookie: <cookie-name>=<cookie-value>;max-age/expires=value;
```

LocalStorage()	SessionStorage()	Cookies
5MB/10MB Storage	5MB Storage	4KB Storage
It's not session based ,need to deleted via JavaScript by using clear method of local storage For ex:- localStorage.clear();	It's session based i.e. working per window or tab	Expiry depends upon setting and working per window and also We can set the expiration time for each cookie
only client side reading support	only client side reading	client side reading and server side reading support
less order browser support	less order browser support	more order browser support

Older*

20.4 Notification API

The Notifications API allows web pages to control the display of system notifications to the end user. These are outside the top-level browsing context viewport, so therefore can be displayed even when the user has switched tabs or moved to a different app.

The API is designed to be compatible with existing notification systems, across different platforms.

In order to create notification, we need an object of Notification which can be created as shown below:

```
new Notification(title,options)
```

User needs to grant permission to website for displaying notifications. In order to request the permission from the user requestPermission() of Notification object can be used as show below:

```
Notification.requestPermission()
```

```
function notify(){
   Notification.requestPermission();
   if(Notification.permission === "default"){
      alert("Please grant permission");
   }
   else {
      var notify = new Notification("New Mail",{body:"You have 1 unread email"});
   }
}
```

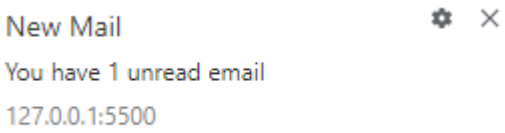
Properties of Notification object:

body:

Defines notification message.

permission:

It is a static property of Notification interface, which represent current permission to display notifications. Possible values for permission are denied,granted,default (the user choice is unknown so that browser will act as if the value is denied)
As a web developer, whenever you are developing a web-application that is supposed to notify user upon receiving updated data, you should use Notification API.



21. Meta- Programming

21.1 Symbols

Symbols are used as unique identifiers especially to identify properties within an object.

The value used for creating a symbol acts like a description and can not be used to access the symbol uniquely.

```
const sym1 = Symbol('book');
console.log(sym1);
```

```
Symbol(book)
```

The symbol variables having same description are internally unique and hence the output of the below snippet on the console will display false.

```
const sym2 = Symbol('book');
const sym3 = Symbol('book');
console.log(sym2 === sym3);
```

```
false
```

Each book has unique id and name, but when we add them in the above format instead of adding 4 books to the collection; everytime, it overrides the book details and retains the last one added.

```
let books = {
  'book': { id: 12345, name: 'Node' },
  'book': { id: 53579, name: 'jQuery' },
  'book': { id: 90687, name: 'Angular' },
  'book': { id: 86542, name: 'Express' }
};
console.log(books);
```

```
▼ {book: {…}}
  ▶ book: {id: 86542, name: "Express"}
  ▶ __proto__: Object
```

```
let books = {
  [Symbol('book')]: { id: 12345, name: 'Node' },
  [Symbol('book')]: { id: 53579, name: 'jQuery' },
  [Symbol('book')]: { id: 90687, name: 'Angular' },
  [Symbol('book')]: { id: 86542, name: 'Express' }
};
console.log(books);
```

Even though the descriptions are same, each Symbol is unique and hence we can now see 4 books.

In ES6, Symbol.iterator is present in all the iterable objects.

21.2 Iterables

The Iterable Protocol

- In ES6 we have the flexibility to choose how a particular object should be iterated.
- For some of the built-ins objects, we have a predefined behavior for iteration. Example: Strings, Arrays etc.
- With iterable protocol, we can define custom way of iteration for a given object.
- The new for…of loop can be used to iterate over any iterable.
- In order for an object to be iterable, it must implement the iterable interface.
- The iterable interface enforces implementation of iterator method which defines how the object should be iterated.
- The iterator method returns an iterator object that conforms to the iterator protocol.

The Iterator Protocol

- The iterator protocol is used to define a standard way that an object produces a sequence of values.
- Using this we can define how an object should iterate and this is implemented using the .next() method.
- An object becomes an iterator when it implements the .next() method that returns an object with two properties:
    - value : the data representing the next value in the sequence of values within the object
    - done : a boolean representing if the iterator is done going through the sequence of values
    - If done is true, then the iterator has reached the end of its sequence of values.
    - If done is false, then the iterator is able to produce another value in its sequence of values.

```
const digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
const arrayIterator = digits[Symbol.iterator]();

console.log(arrayIterator.next());
console.log(arrayIterator.next());
console.log(arrayIterator.next());
```

```
▶ {value: 0, done: false}
▶ {value: 1, done: false}
▶ {value: 2, done: false}
```

21.3 Generators

Function in most JavaScript code follow a single threaded approach and can not be interrupted once they start.
Generators are special types of functions which can be paused.
The new syntax is introduced in ES6 for generator function as below:

```
function* myFunction() {
    //some code here
}
```

Pausable Functions

Adding an * symbol between the function keyword and the name makes it a generator function.

Generators and Iterators

When a generator is invoked, it doesn't actually run any of the code inside the function. Instead, it creates and returns an iterator. This iterator can then be used to execute the actual generator's inner code.

Calling next() resumes the suspended function execution and gets the previous output.

```
const generatorIterator = getEmployee();
generatorIterator.next();
```

Pausable Function with Generators

Using the new “Yield” keyword :

The yield keyword is new and was introduced with ES6. It can only be used inside generator functions. yield is what causes the generator to pause.

```
function *getEmployee(){
    console.log('the function has started');

    const names = ['Subhash', 'Khalid', 'Ravi',
    'Dhanya', 'Prajila', 'Kavya', 'Vidya', 'Veena'];

    for (const name of names) {
        console.log(name);
        yield;
    }

    console.log('the function has ended');
}
```

```
getEmployee();
▶ getEmployee {<suspended>}
const generatorIterator = getEmployee();
generatorIterator.next();
the function has started
Subhash
▶ {value: undefined, done: false}
generatorIterator.next();
Khalid
▶ {value: undefined, done: false}
generatorIterator.next();
Ravi
▶ {value: undefined, done: false}
```

The yield causes the function to pause after printing each name on console. The next() causes it to resume

```
function *getEmployee() {
    console.log('the function has started');

    const names = ['Subhash', 'Khalid', 'Ravi',
    'Dhanya', 'Prajila', 'Kavya', 'Vidya', 'Veena'];

    for (const name of names) {
        yield name;
    }

    console.log('the function has ended');
}
```

The yield in this case sends the data out of generator before pausing

```
getEmployee();
▶ getEmployee {<suspended>}

const generatorIterator = getEmployee();
let data = generatorIterator.next();
data.value;
the function has started
"Subhash"
generatorIterator.next().value
"Khalid"
generatorIterator.next().value
"Ravi"
```

21.4 Reflect API

- Object reflection is an ability of a language to inspect and manipulate object properties at runtime.
- JavaScript had been supporting APIs for object reflection but these APIs were not organized under a namespace and also failed to handle exceptions.
- ES2015 introduces a new object known as Reflect which exposes methods for object reflection.
- These methods well manage the error conditions which makes developers easy to write code involving object reflection.
- The concept of reflection is very similar to the one in higher programming languages like Python and C#.

21.4.1 Construct an Object

Reflect.construct(Date, [1982, 9, 22]);

```
new Date("22-Oct-1982")
Fri Oct 22 1982 00:00:00 GMT+0530 (India Standard Time)
Reflect.construct(Date, [1982, 9, 22]);
Fri Oct 22 1982 00:00:00 GMT+0530 (India Standard Time)
```

21.4.2 Define Property

The Reflect.defineProperty method allows precise addition to or modification of a property on an object. Object.defineProperty returns the object or throws a TypeError if the property has not been successfully defined.

Reflect.defineProperty, however, simply returns a Boolean indicating whether or not the property was successfully defined.

Reflect.defineProperty(target, propertyKey, attributes)

Parameters

- target - The target object on which to define the property.
- propertyKey - The name of the property to be defined or modified.
- attributes - The attributes for the property being defined or modified.

```
let obj = {};
Reflect.defineProperty(obj, 'x', {value: 7}); // true
obj.x; //7
7
```

21.4.3 Delete Property

The Reflect.deleteProperty method allows you to delete a property on an object. It returns a Boolean indicating whether or not the property was successfully deleted. It is almost identical to the non-strict delete operator.

Reflect.deleteProperty(target, propertyKey)

Parameters

- target - The target object on which to delete the property.
- propertyKey - The name of the property to be deleted.
- Return value - A Boolean indicating whether or not the property was successfully deleted.

```
let obj = { x: 3, y: 4 };
Reflect.deleteProperty(obj, 'x'); // true
obj; // { y: 4 }
▶ {y: 4}
let arr = ['a', 'b', 'c', 'd', 'e'];
Reflect.deleteProperty(arr, '3'); // true
arr; // [a, b, c, , e]
▶ (5) ["a", "b", "c", empty, "e"]
// Returns true if no such property exists
Reflect.deleteProperty({}, 'foo'); // true
true
// Returns false if a property is unconfigurable
Reflect.deleteProperty(Object.freeze({foo: 'x'}), 'foo'); // false
false
```

21.4.4 get Property

The Reflect.get method allows you to get a property on an object.

```
Reflect.get(target, propertyKey, receiver)
```

Parameters

- target - The target object on which to get the property.
- propertyKey - The name of the property to get.
- receiver (optional) - The value of this provided for the call to target if a getter is encountered.

```
// Object
let obj = { x: 1, y: 'hello' };
Reflect.get(obj, 'y'); // hello
```

```
"hello"
```

```
// Array
Reflect.get(['one', 'two'], 1); // "two"
```

```
"two"
```

```
// Proxy with a get handler
let x = {p: 1};
let myobj = new Proxy(x, {
get(t, k, r) { return k + 'bar'; }
});
Reflect.get(myobj, 'foo'); // "foobar"
```

```
"foobar"
```

21.5 Proxy API

- Proxies are created for intercepting an object property.
- As the name indicates it creates a proxy object for a given object and an additional handler to it.
- To create a proxy object, we use the Proxy constructor - new Proxy();.
- The proxy constructor takes two parameters:
    - the object that it will be the proxy for.
    - an object containing the list of methods it will handle for the proxied object. The second object is called the handler.

21.5.1 Get Trap

The get trap is invoked whenever any property of the target is accessed.

The get trap receives the target object and the property name as parameters.

```
var myObject = {status: 'Creating ES6 Course Content'};
const handler = {
    get(target, propName) {
        console.log(target);
        console.log(propName);
        return target[propName];
    }
};
var reviewer = new Proxy(myObject, handler);
reviewer.status;
```

```
▶ {status: "Creating ES6 Course Content"}
status
"Creating ES6 Course Content"
```

Property value returned for target[propName]

21.5.2 Set Trap

The set trap is used as an interceptor whenever there is a value change in a property.

The set trap receives the object it proxies, the property that is being set and the new value for the proxy.

```
var myObject = {status: 'Creating ES6 Course Content',completion:90};
const handler = {
    set(target, propName, value) {
        if (propName === 'completion' && value == 100) {
            target['status']='Courseware completed';
        }
        target[propName]=value;
    }
};
var reviewer = new Proxy(myObject, handler);
reviewer.completion=98;
reviewer.status;
"Creating ES6 Course Content"
```

The set trap helps in appropriately setting the value to the current as well as any property of the target object based on the conditional statements.

```
reviewer.completion=100;
reviewer.status;
"Courseware completed"
```

22. JavaScript Security

JavaScript security refers to the security measures that every JavaScript developer should follow while developing web applications. JavaScript security encompasses measures to be practiced to improve web application safety by identifying and preventing the weaknesses in the application.

As we all know that JavaScript is the most popular language used for web application development and the way JavaScript interacts with DOM(Document Object Model) causes a lot of risk for end-users, by allowing attackers to get into the code easily by finding the vulnerable component in the application.

Also, JavaScript can be executed outside the browser through the Node.js platform (runtime environment for developing server-side of the application) makes it more important to implement security features in JavaScript applications.

There are many JavaScript-based frameworks available now which provide access to the native system APIs, which offers a lot of flexibility to JavaScript developers, and also, on the other hand, this makes JavaScript vulnerable to different types of attacks. Attackers can gain unauthorized access to the application in the absence of safe coding practices in the application.

Hence, by following the security measures we can write secure JavaScript code.

22.1 Security Challenges in JavaScript

Most of the attacks in JavaScript occurs by exploiting vulnerabilities present in web applications. An attacker may inject malicious JavaScript code into an existing file or edit the HTML content of the web application and invoke a third-party JavaScript file that has the malicious code.

One of the prominent security issues in webpages on the internet is malicious JavaScript code and it creates a severe impact as well. When comparing to different types of network attacks, it is tough to detect malicious JavaScript code as an attacker can inject malicious code in different ways. Also, the code will get interpreted by the browser immediately. This allows JavaScript attacks to exploit many types of weaknesses in the browser's environment.

Consider the below realities to understand the probability of JavaScript being malicious:

- JavaScript can access user's sensitive information like cookies
- JavaScript can send HTTP requests to different destinations using XMLHttpRequest or any other tool
- JavaScript can transform the HTML content of a web page using different DOM(Document Object Model) manipulation methods

Severe security breaches may take place in the web application because of these facts. If an attacker can inject illogical JavaScript into a web page served by a website, then the security of the website and the users visiting that website have been compromised.

The attacker can accomplish the diverse types of attacks by executing illogical JavaScript in the user's browser.

- Keylogging: The hacker can send the user's keystrokes to his server, and henceforth record sensitive information such as passwords and credit card numbers.
- Cookie theft: The hacker can access the target user's cookie associated with the website and thus extract sensitive information using it.
- Phishing: The hacker can insert a fake login form into the target's page and then get the form data submitted to his server and acquire sensitive information like a password.

Language level security aspects

Let us look at some of the JavaScript specific security issues.

1. JavaScript is inherently a global language

Variables in JavaScript are in the global namespace by default. The variables which are in the global scope can lead to security flaws. Functions that are in the global scope are also dangerous.

For example: Let us consider the below global function:

```
var emp = {
        name: 'RM',
        details:function(){
                alert(emp.name);
        }
};
```

The global function can be called from the browser and attackers might find a way to enter into the code. There are several approaches to avoid this and one of the most popular ones is revealing module pattern.

2. ES6 (ECMAScript) added support for template literals, in general, which are very useful. However, this also allows attackers to use this functionality to bypass protection measures against the very common attack and leads to a dangerous attack called cross-site scripting(XSS).

```
template= `Converted:${alert("converted to alert")}`;
```

In the above code, ${} evaluates the content and generates JavaScript dynamic code alert() with value as shown below:

Here, if the user provides malicious input then that gets executed as part of this code. Hence, as a security measure, it is good practice to sanitize and validate user input before using it in processing.

3. Should not use JavaScript's eval() function:

The eval () function executes or evaluates an expression as a code. It is more dangerous when it is invoked on a string that is constructed partially from user input. Also improper usage of eval() opens your website to injection attacks.

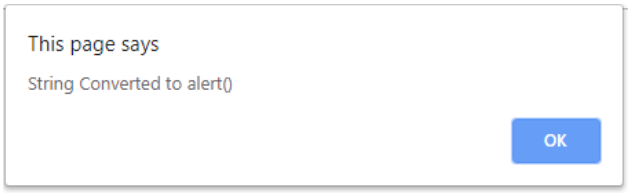
```
console.log('al' +  'ert(\" + 'String Converted to alert()' + '\')');
```

The above line logs the content alert('String Converted to alert()') to the console.

But when we pass the same content as an argument to the eval() function as follows:

```
eval('al' +  'ert(\" + 'String Converted to alert()' + '\')');
```

The above string gets converted to JavaScript code, hence it alerts with the content String Converted to alert() as follows:



4. JavaScript is a weakly typed language: We do not have an explicit strict type for a particular variable in JavaScript.

22.2 Cross-Site Scripting (XSS)

XSS is a vulnerability in web applications and it can also be defined as an attack on the client-side in which the attacker injects and runs a malicious script into a web page. Attackers regularly use diverse channels like message boards, comments, or forums to carry wicked scripts to the user's browser.
A web application is susceptible to XSS if it uses unsanitized user input in the response which is interpreted by the browser. XSS attacks are possible in VBScript, ActiveX, Flash, and even CSS. However, they are mostly seen in JavaScript, mainly because JavaScript is ultimate to most browsing experiences.
The cross-site scripting attack can also be defined as a client-side code injection attack because any code executed by the web browser after loading the page is normally designated as the client-side code.

One of the common XSS attacks is seen in websites that have comment forums that are not validated. The attacker will post a comment into the forum with malicious code wrapped in <script> tag. The browser will interpret the code between the <script> tag as valid JavaScript code and when any other user access the website, the malicious code between the script tags will get executed. Thus the user became victim to this attack.

Actors in an XSS attack

An XSS attack involves mainly three actors:

- The Website: responsible for serving HTML pages as a response when users request them.
- The Victim: the website users who send requests from their browser
- The Attacker: the malicious user of the website who attacks the victim.

Injecting unsanitized input into a web application results in the XSS attack. Some of the options to inject XSS attack into a web application are below:

- <script> tag: The <script> tag is the one of the direct XSS payload. It is possible to reference an external JavaScript code within the <script> tag or JavaScript code can be directly embedded within the <script> tag.
- JavaScript events: Another common XSS attack vector is event attributes. The event attributes such as "onload" and "onerror" can be used in diverse tags.

```
<b onmouseover=alert('Test!')>click me!</b>
```

- <body> tag: It is possible to deliver an XSS payload inside the <body> tag by using event attribute or other attributes like background attribute.

```
<body onload=alert('test')>
```

- <img> tag: Ths XSS attack can happen on executing the JavaScript code found in the <img> attributes as well.
- <input> tag: On setting the type attribute of the <input> tag as an image, then it is possible to manipulate it and embed the script.

Types of XSS attack

22.2.1 DOM XSS

Each HTML document will have a DOM (Document Object Model) connected with it which comprises objects that signify the properties of that document. Any client-side script execution can use the DOM of the HTML page and the script can access the properties of that page and be able to change the values accordingly.

DOM XSS is a variety of cross-site scripting weakness that executes in the DOM instead of part of the HTML. DOM-based XSS is also known as Type-0 XSS. It is an XSS attack where the attacker modifies the victim's browser's DOM environment and thereby modifying it and thus the client-side code executes in an unexpected way.

Whenever the web application writes data to the DOM without proper sanitization, it is possible for a DOM-based XSS attack. The attacker can inject malicious JavaScript code to manipulate the data on the web page.

Sources: The assets read from the DOM is named as a source. To achieve a DOM XSS attack, a script can be injected into this asset.

The below table gives some of the DOM properties that are frequently misused for DOM-based XSS attacks. Some of the best-known sources that are generally misused for DOM-based XSS are:

Source name	Property
document	document.URL document.documentURI
location	location.href location.search
Referrer	document.referrer
Window Name	window.name

Sinks: The points in data flow at which the untrusted input gets executed by JavaScript within the page is named as a sink. Some of the best-known sinks that are generally misused for DOM-based XSS are:

Sink Name	Property
HTML element Sink	document.write document.writeIn innerHTML outerHTML
Execution Sink	eval setTimeout setInterval
Set Location sink	location location.href

Some of the preventive measures are:

- Prevent the user from clicking any suspicious links
- Keep the browser up to date
- Use JavaScript framework: The latest JavaScript frameworks like Angular and libraries like React uses templates that checks for XSS attacks and do input sanitization.
- Do code auditing: Avoid using the properties and functions like innerHTML, outerHTML, document.write, and better to use innerText, or textContent properties to set text content within the tag.

22.2.2 Reflected XSS

Reflected Cross-site Scripting(XSS) also known as Non-persistent Cross-site Scripting or Non-persistent XSS is a type of XSS where an attacker injects browser executable code through input which is sent back within the HTTP response. The injected code is non-persistent and not stored in the application.

The attack payload gets delivered and executed through a single request and response and hence Reflected XSS is also denoted as Type 1 XSS.

The attacker searches for vulnerable websites in order to attack the site. Once a site is identified as vulnerable, then the attacker will try to inject the script and verify whether the code is returned as such.

The core aim of this kind of attack:

- Cookie stealing – attackers steal cookies information and perform extra attacks.
- Data-stealing – attackers read file contents, browser history, directory listings, and plan for further attacks.

Defending against this attack can be done on the client-side or server-side.

1. Client-side: Some preventive measures on the client-side to defend against this attack are:

- Open the emails which come from a trusted source only
- Avoid installing browser plugins unnecessarily, use only secure browsers
- Do proper sanitization and validation on user input

2. Server-side: Some preventive measures on the server-side to defend against this attack are:

- Do proper validation of user input before sending user data back in the response.
- Sanitize, escape or encode the special chars in user input before sending it back in an HTTP response. This will make sure that the script is rendered harmless.

22.2.3 Persistent XSS

The Persistent XSS attack is a variant of cross-site scripting. It is also known as stored XSS. It happens when the server stores the data injected by the attacker into the web page, and then the server displays the data to other users who are surfing the website as well. This injected data could be stored in a file, database, cookie, or even as session data. On visiting the vulnerable web page, the payload gets served to the victim and get executed in their environment. Most of the users, who are visiting the vulnerable web page will get infected as malicious data is persisted in the server.

Persistent XSS attacks are happening out through two requests and so it is also known as Type 2 XSS attacks. The first request specifies the mean code injection and its persistence on the webserver. The second request specifies target victims loading the pages that have the payload.

22.3 Cross-Site Request Forgery (CSRF)

CSRF (Cross-Site Request Forgery) is an attack where an attacker will do undesirable actions in an application for the actual user who logged in. This attack targets functionality of state-changing requests such as changing email, password or purchasing, etc., as an attacker will not be able to fetch data from the response.

As for most web applications, browser requests hold information such as the user's session cookie, IP address, etc., it is possible for an attacker to fetch the privileges of an application. Thus, if a user has logged into a web application, the application will not be able to differentiate between the fake request and the genuine request sent by the user. Due to this, it will be easier for an attacker to trick the user to execute unwanted actions.

This attack is also known by other names like XSRF, Sea Surf, Session Riding, Hostile Linking, and One-click attack.

To execute this attack, an attacker will first study the application in order to understand how to make a forged request appear as a legitimate request. There are two means to fake a user.

1. Sending emails or clicking links

In this, the attacker will send some malicious links or emails to the user's web application and makes a user click it for the attack to execute.

2. Sending a request which looks like a legitimate request

In this, the attacker will be sending a request which includes cookies associated with the user's session. Any request which is sent with these cookies will be considered as a genuine one as at any time a request is made to a web application, the user's browser will check if it has any cookies linked with it and that needs to be directed along with HTTP request. If so all cookies will be included in all requests to that web application. If the web application approves the session cookie, an attacker can use CSRF to send requests as if the actual user is sending them. The web application will not be capable to discriminate between requests sent by the user and attacker as every request will be sent from the user's browser with their own cookie.

22.3.1 CSRF Mitigation Techniques

The most popular and recommended mitigation technique for a CSRF attack is using anti-csrf tokens.

We need to send a secure random CSRF token for each state-changing request to prevent CSRFattack. A CSRF token should be generated using a cryptographic random generator and should be unique for each user session. The CSRF token is added as a hidden field for forms and as a parameter for Ajax calls.

The most secure way of adding a CSRF token as a header in the HTTP request is via Javascript than adding it as a hidden field. Most of the frameworks like Spring, .Net, etc., have in-built support for CSRF support. If you are using any of these frameworks, better to make use of the built-in methods instead of generating the token manually.

Let us now see how an anti-csrf token is generated using the Express framework.

csurf is a Node module available to generate csrf tokens to mitigate the CSRF attack. Install it along with the cookie-parser node module inside your project folder as shown below:

```
npm install csurf --save
npm install cookie-parser –save
```

As this token needs to be generated from the Express framework, in this case, we need to configure the following statement in Express code:

```
const csrfMiddleware = csrf({ cookie: true });
```

Then add this as a middleware for requests as shown below:

```
app.get('/api/getToken', csrfMiddleware, function (req, res) {
   res.json({ _csrf: req.csrfToken() });
});
```

Here req.csrfToken() generates a random token and will be sent as a response for a get request. Now the client has to store it in a hidden field so that it is sent along with every request back to the server for validation.Below is the code snippet for the same in the client code.

```
<form action='http://localhost:1000/changePassword' method='POST'>
    <label> Enter new Password : </label>
    <input type='password' name='password' id='' />
    <input type='hidden' value="_csrf" name="_csrf" id="_csrf" />
    <button type='submit'>Submit</button>
</form>
Instead of using a hidden field, it's better to send a CSRF token using x-csrf-token.
```

CSRF Mitigation Myths

There are some more mitigation techniques that are assumed to prevent this attack, but in reality, they do not prevent it.

- CORS: This is a header that is used for cross-domain communication between the applications. This will not prevent CSRF attacks.
- HTTPS: Using HTTPS also doesn't prevent CSRF attacks. Resources using HTTPS are still vulnerable to CSRF attacks.

Best practices to mitigate CSRF attacks

- Logoff from the application when not in use
- Do not save username/password in a browser and do not allow the browser to remember logins
- Do not use the same browser to access sensitive applications and to surf websites on the internet

23. JavaScript Testing

Unit testing involves the execution of a JavaScript function to evaluate if the output returned is the expected one. It helps us test if one or more features of a JavaScript function are working as per expectation. In general, these features indicate the extent to which the JavaScript application being tested:

- meets the requirements that guided its design and development,
- responds correctly to all kinds of inputs,
- achieves the general result its stakeholders desire

To understand the features of Jasmine framework and to use it to solve the actual requirement, let us consider a simple function totalTravelFare(baseFare, taxPercentage) as shown below:

```
function totalTravelFare(baseFare, taxPercentage){
        var finalFare;
        finalFare = baseFare*(1+taxPercentage/100);
        return finalFare;
}
```

Step 1: A test suite should be created

A test suite is a grouping of relevant test cases which are executed together. We can create a test suite with the help of built-in function describe().

describe(title, function(){})

- It is a global Jasmine function
- It helps in defining the test context by creating a new test suite

It accepts two parameters:

- A title (of string type) or name of the test suite
- A function containing specs which belong to this suite

For example, we can create a test suite for totalTravelFare as shown below.

```
describe('TotalTravelFare calculation Suite:',function(){

});
```

Step 2: Test specs should be created for all the possible scenarios

A test spec is the actual test case. Now we will write spec inside the suite, with the help of Jasmine function "it".

it()

- It is also a global Jasmine function
- It helps translate the acceptance criteria into Jasmine spec

It accepts two parameters:

- Title or name of the test suite
- Function containing spec code

For example, we can create a test spec for totalTravelFare as shown below.

```
describe('TotalTravelFare calculation Suite:',function(){
    it('Test Case 1: Inputs are correct',function(){
      expect(totalTravelFare(1000,20)).toEqual(1200);
    });
})
```

expect()

- It is a global Jasmine function that helps in writing the assertions.
- It takes only one parameter: Actual value to be tested

toEqual()

- Matcher is used to compare the actual and expected output

Method	Description
toBe (expected)	expect the actual value to be === to the expected value.
toBeDefined ()	expect the actual value to be defined. (Not undefined)
toBeFalsy ()	expect the actual value to be false
toBeTruthy ()	expect the actual value to be true.
toBeGreaterThan (expected)	expect the actual value to be greater than the expected value.
toBeLessThan (expected)	expect the actual value to be less than the expected value.
toEqual (expected)	expect the actual value to be equal to the expected, using deep equality comparison
toMatch (expected)	expect the actual value to match a regular expression
toThrow (expected)	expect a function to throw something.

Step 3: Test suite should be executed with the help of Karma

So far we have just written some test cases with .js extension. To run these test cases we need to install jasmine and karma. Karma is a test runner tool. These tools need to be downloaded and installed. Node has a something called npm (Node Package Manager) which helps us install such tools.

Type the below commands in the vscode terminal:

```
npm install -g karma jasmine-core
```

Karma is:

A tool that spawns a web server which executes the source code against the test code for each browsers connected.
When executed, it automatically captures the browser specified by the developer during Karma configuration.
It then displays the results on the command line.
It watches all the files specified within configuration file and if there are any changes, it will trigger the corresponding spec again on the browser.