JavaScript ES6 Features

1) The let keyword
2) The const keyword
3) Arrow Functions
4) The ... Operator
5) For/of
6) Map Objects
7) Set Objects
8) Classes
9) Promises
10) Symbol
11) Default Parameters
12) Function Rest Parameter
13) String.includes()
14) String.startsWith()
15) String.endsWith()
16) Array.from()
17) Array keys()
18) Array find()
19) Array findIndex()
20) New Math Methods
21) New Number Properties
22) New Number Methods
23) New Global Methods
24) Object entries
25) JavaScript Modules

Features in Details

The let keyword allows to declare a variable with block scope.

```
var x = 10;
// Here x is 10
{
  let x = 2;
  // Here x is 2
}
// Here x is 10
```

The const keyword allows to declare a constant (a JavaScript variable with a constant value).
Constants are similar to let variables, except that the value cannot be changed.

```
var x = 10;
// Here x is 10
{
  const x = 2;
  // Here x is 2
}
// Here x is 10
```

The Arrow Functions
Arrow functions allows a short syntax for writing function expressions.
No need of using the function keyword, the return keyword, and the curly brackets.

```
// ES5
var y = function()(
    return 100;
}

var a = function(x, y) {
   return x * y;
}

// ES6

const y = () = > 100;

const a = (x, y) => x * y;

const z  () => {
  console.log('hello');
  console.log('world');
}
```

Arrow functions do not have their own **this**. They are not well suited for defining object methods.
Let variables and Arrow functions are not hoisted. They must be defined before they are used.

Hoisting is the default behavior in JavaScript where declarations of variables and functions are moved to the top of their respective scopes during the compilation phase. This ensures that regardless of where variables and functions are declared within a scope, they are accessible throughout that scope.

Using **const** is safer than using var, because a function expression is always a constant value.
You can only omit the **return** keyword and the curly brackets if the function is a single statement. Because of this, it might be a good habit to always keep like:

```
const a = (x, y) => { return x * y };

const greeting = () => {
  return 'hello world';
}
```

```
greeting() => 'hello world';
```

The arrow function expression can be storeed inside a variable or constant or use it as a parameter where callback function is passed.

```
let data = () => {
  console.log('hello');
  console.log('world');
}
```

use as: data();
btn.onClick = data;

The Spread (...) Operator
The ... operator expands an iterable (like an array) into more elements:

```
const quarter1 = ["Jan", "Feb", "Mar"];
const quarter2 = ["Apr", "May", "Jun"];
const quarter3= ["Jul", "Aug", "Sep"];
const quarter4 = ["Oct", "Nov", "May"];

const year = [...quarter1, ...quarter1, ...quarter3, ...quarter4];
```

The ... operator is also used to expand an iterable into more arguments for function calls:

```
const numbers = [23,55,21,87,56];
let minValue = Math.min(...numbers);
```

The For/Of Loop
The JavaScript for/of statement loops through the values of an iterable objects.
for/of lets to loop over data structures that are iterable such as Arrays, Strings, Maps, NodeLists and more.

The for/of loop has the following syntax:

```
for (variable of iterable) {

  // code block to be executed

}
```

variable - For every iteration the value of the next property is assigned to the variable. Variable can be declared with const, let, or var.
iterable - An object that has iterable properties.

```
const cars = ["BMW", "Volvo", "Maruti"];
let models = "";

for (let name of cars) {
  models += name + " ";
}
```

Looping over a String

```
let language = "JavaScript";
let text = "";

for (let x of language) {
   text += x + " ";
}
```

JavaScript Maps
Use an Object as a key in Map.

```
const fruits = new Map([
["apples", 500],
["bananas", 300],
["oranges", 200]
]);

var count = fruits.get("bananas")
```

JavaScript Sets

```
// Create a Set
const letters = new Set();

// Add some values to the Set
letters.add("a");
letters.add("b");
letters.add("c");
var length = letters.size;
```

JavaScript Classes
JavaScript Classes are templates for JavaScript Objects.
Use the keyword class to create a class.
Always add a method named constructor():

```
class Car {
  constructor(name, year) {
    this.model = name;
    this.mfg = year;
  }
}
```

creates a class named "Car".
The class has two initial properties: "model" and "mfg".
A JavaScript class is not an object.
It is a template for JavaScript objects.

use the class to create objects:

```
const Car1 = new Car("Ford", 2014);
const Car2 = new Car("Audi", 2019);
consoe.log(car1.model);
console.log(car2.mfg);
```

Private Methods and Fields

```
class User {
  #counter = 0;  // Private field
  #update() {} // Private method
}
const data = new User();

let x = data.#counter; // Error
data.#update();      // Error
```

JavaScript Promises
A Promise is a JavaScript object that links "Producing Code" and "Consuming Code".
"Producing Code" can take some time and "Consuming Code" must wait for the result.

With Promise the Consuming code need not have to wait till the Producing code is finished the execution.

Consuming code can continue doing other activities after calling the Producing code (which returns a promise) and whenever the Producing code returns sucessfully, it will notify the Consuming code via promise.
If the Consuming wishes to check the status, it can check wherther the Producing code has finished or not by looking at the promise.
The Promise the link object that links "Producing Code" and "Consuming Code" in Async execution of Producing code.

```
const prms = new Promise(function(resolver, rejector) {
  setTimeout(function() { myResolve("I love You !!"); }, 3000);

//returns after a delay of 3000 msec.
 if(true)
      rejector(“errr!”);
});
```

The resolver and rejector are callback functions.

```
prms.then(function(value) {
  //document.getElementById("demo").innerHTML = value;
});
```

The Symbol Type

A JavaScript Symbol is a primitive data type just like Number, String, or Boolean.

It represents a unique "hidden" identifier that no other code can accidentally access.

For instance, if different coders want to add a person.id property to a person object belonging to a third-party code, they could mix each others values.

Using Symbol() to create a unique identifiers, solves this problem:

```
const person = {
  firstName: "Ashoka",
  lastName: "Mourya",
  age: 50,
  eyeColor: "red"
};

let id = Symbol('id');
person[id] = 14045;
// Now person[id] =  14045
// but person.id is still undefined
```

Symbols are always unique.If you create two symbols with the same description they will have different values:

```
Symbol("id") == Symbol("id"); // false
```

Function with Default Parameter Values

With ES6 function parameters can have default values.

```
function  getSum(a, b = 10) {

  // y is 10 if not passed or undefined

  return x + y;

}

getSum(5); // will return 15

getSum(5,20) //25
```

Function with rest Parameter (variabl numbre of parameters)

The rest parameter (...) allows a function to treat an indefinite number of arguments as an array:

```
function getSum(...args) {
  let sum = 0;
  for (let arg of args) {
     sum += arg;
     return sum;
}
var total = sum(4, 9, 16, 25, 29, 100, 66, 77,22);
```

Trailing Commas

JavaScript allows trailing commas wherever a comma-separated list of values is accepted.

In Array and Object Literals, Function Calls, Parameters, Imports and Exports.

```
function update(x,,,) {};
const arrayData = [1,2,3,4,,,];
const userObj = {fname: baba, age:50,,,};
```

Object Rest Properties

ECMAScript 2018 added rest properties.

This allows to destruct an object and collect the leftovers onto a new object:

```
let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
x; // 1
y; // 2
z; // { a: 3, b: 4 }
```

Template Literals

ES6 introduces very simple string templates along with placeholders for the variables.

The syntax for using the string template is ${PARAMETER} and is used inside of the back-ticked string.

```
let name = `My name is ${firstName} ${lastName}`
```

Destructuring Assignment

Array Destructuring

```
let fruits = ["Apple", "Banana"];
let [a, b] = fruits; // Array destructuring assignment
console.log(a, b);
```

Object Destructuring

```
let person = {name: "baba", age: 28};
let {name, age} = person; // Object destructuring assignment
console.log(name, age);
```

Enhanced Object Literals

ES6 provides enhanced object literals which make it easy to quickly create objects with properties inside the curly braces.

```
function getModel(manufacturer, model, year) {
  return {
    manufacturer,
    model,
    year
  }
}
getModel("Samsung", "Galaxy", "2020");
```

Multi-line Strings.

Ceate multi-line strings by using back-ticks(`).

```
let greeting = `Hello World,
          Greetings to all,
          Keep Learning and Practicing!`
```

String.includes()

The includes() method returns true if a string contains a specified value, otherwise false:

```
let text = "Hello world, welcome to the universe.";

text.includes("world")    // Returns true
```

String.startsWith()

The startsWith() method returns true if a string begins with a specified value, otherwise false:

```
let text = "Hello world, welcome to the universe.";

text.startsWith("Hello")   // Returns true
```

String.endsWith()

The endsWith() method returns true if a string ends with a specified value, otherwise false:

```
var text = "John Doe";

text.endsWith("Doe")    // Returns true
```

Array.from()

The Array.from() method returns an Array object from any object with a length property or any iterable object.

Create an Array from a String:

```
Array.from("ABCDEFG")   // Returns [A,B,C,D,E,F,G]
```

Array keys()

The keys() method returns an Array Iterator object with the keys of an array.

Create an Array Iterator object, containing the keys of the array:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];

const keys = fruits.keys();

let text = "";

for (let x of keys) {

  text += x + "<br>";

}
```

Array find()

The find() method returns the value of the first array element that passes a test function.

This example finds (returns the value of) the first element that is larger than 18:

```
 const numbers = [4, 9, 16, 25, 29];
let first = numbers.find(search); //callback function
function search(value, index, array) {
  return value > 18;
}
```

Array findIndex()

The findIndex() method returns the index of the first array element that passes a test function.

This example finds the index of the first element that is larger than 18:

```
 const numbers = [4, 9, 16, 25, 29];
let first = numbers.findIndex(serachIndex);
function serachIndex(value, index, array) {
  return value > 18;
}
```

New Math Methods

ES6 added the following methods to the Math object:

Math.trunc()

Math.sign()

Math.cbrt()

Math.log2()

Math.log10()

N**ew Number Properties**

ES6 added the following properties to the Number object:

EPSILON

MIN_SAFE_INTEGER

MAX_SAFE_INTEGER

EPSILON usage

let x = Number.EPSILON;

MIN_SAFE_INTEGER

let x = Number.MIN_SAFE_INTEGER;

MAX_SAFE_INTEGER

let x = Number.MAX_SAFE_INTEGER;

New Number Methods

ES6 added 2 new methods to the Number object:

Number.isInteger()

Number.isSafeInteger()

The Number.isInteger() Method

The Number.isInteger() method returns true if the argument is an integer.

```
Number.isInteger(10);        // returns true
Number.isInteger(10.5);      // returns false
```

Safe integers are all integers from -(253 - 1) to +(253 - 1).

This is safe: 9007199254740991. This is not safe: 9007199254740992.

The isFinite() Method

The global isFinite() method returns false if the argument is Infinity or NaN.

Otherwise it returns true:

```
isFinite(10/0);       // returns false
isFinite(10/1);       // returns true
```

The isNaN() Method

The global isNaN() method returns true if the argument is NaN. Otherwise it returns false:

```
isNaN("Hello");       // returns true
```

Object entries()

Create an Array Iterator, and then iterate over the key/value pairs:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
const f = fruits.entries();
for (let x of f) {
  document.getElementById("demo").innerHTML += x;
  }
```

The entries() method returns an Array Iterator object with key/value pairs:

```
[0, "Banana"]
[1, "Orange"]
[2, "Apple"]
[3, "Mango"]
```

The entries() method does not change the original array.

JavaScript Object Values

Object.values() are similar to Object.entries(), but returns a single dimension array of the object values:

```
const person = {
  firstName : "John",
  lastName : "Doe",
  age : 50,
  eyeColor : "blue"
};

let text = Object.values(person);
```

JavaScript Modules

JavaScript modules allow you to break up your code into separate files.

This makes it easier to maintain a code-base.

Modules are imported from external files with the import statement.

Modules also rely on type="module" in the <script> tag.

Example

```
<script type="module">
import message from "./message.js";
</script>
```

Named Exports

Let us create a file named person.js, and fill it with the things we want to export.

You can create named exports two ways. In-line individually, or all at once at the bottom.

In-line export individually:

person.js

```
export const name = "Jesse";
export const age = 40;
```

Export all at once at the bottom:

person.js

```
const name = "Jesse";
const age = 40;
export {name, age};
```

Default Exports

Let us create another file, named message.js, and use it for demonstrating default export.

You can only have one default export in a file.

Example

message.js

```
const message = () => {
```

```
const name = "Jesse";
const age = 40;
return name + ' is ' + age + 'years old.';
};
export default message;
```

Modules only work with the HTTP(s) protocol.

A web-page opened via the file:// protocol cannot use import / export.

Module Import

Modules are imported in two different ways:

Import from named exports

Import named exports from the file person.js:

```
import { name, age } from "./person.js";
```

Import from default exports

Import a default export from the file message.js:

```
import message from "./message.js";
```

JavaScript Async Functions

Defined for calling functions in anon-blocking manner

Waiting for a Timeout

```
async function asyncDisplay() {
  let myPromise = new Promise(function(myResolve, myReject) {
    setTimeout(function() { myResolve("There you are !!"); }, 3000);
  });
  document.getElementById("demo").innerHTML = await myPromise;
}
asyncDisplay();
```

await import

JavasSript modules can now wait for resources that require import before running:

```
import {myData} from './myData.js';

const data = await myData();
```