Advanced Javascript

ES6 or ECMAScript 6

ES6 or ECMAScript 6 is a scripting language specification which is standardized by ECMAScript International.

The full form of ECMA is European Computer Manufacturer's Association. ECMAScript is a Standard for scripting languages such as JavaScript, JScript, etc. It is a trademark scripting language specification. JavaScript is a language based on ECMAScript. Ie JavaScript is considered as one of the most popular implementations of ECMAScript.

ECMAScript is generally used for client-side scripting, and it is also used for writing server applications and services by using Node.js.

ES6 allows you to write the code in such a way that makes your code more modern and readable. By using ES6 features, we write less and do more, so the term 'Write less, do more' suits ES6.

Function Expressions

In Javascript, functions can also be defined as expressions. For example,

```
// program to find the square of a number
let x = function (num) { return num * num };
alert(x(4));

// can be used as variable value for other variables
let y = x(3);
alert(y);
```

Output

```
16
9
```

In the above program, variable x is used to store the function. Here the function is treated as an expression. And the function is called using the variable name.

The function above is called an anonymous function.

JavaScript expressions are written as arrow functions

Arrow Function

Arrow function allows you to create functions in a cleaner way compared to regular functions. For example,

```
// function expression
let x = function(a, b) {
  return a * b;
}
```

can be written as

```
// using arrow functions
let x = (a, b) => a * b;
```

using an arrow function.

```
alert(x(5,5));
```

output 25

Arrow Function Syntax

The syntax of the arrow function is:

```
let myFunction = (arg1, arg2, ...argN) => {
   statement(s)
}
```

Here,

myFunction is the name of the function

arg1, arg2, ...argN are the function arguments

statement(s) is the function body

function as:

let myFunction = (arg1, arg2, ...argN) => expression

Arrow Function with No Argument

If a function doesn't take any argument, then you should use empty parentheses. For example,

```
let greet = () => alert ('Hello');
greet(); // Hello
```

Arrow Function with One Argument

If a function has only one argument, you can omit the parentheses.

For example,

```
let greet = x => alert (x);
greet('Hello'); // Hello
```

Arrow Function as an Expression

You can also dynamically create a function and use it as an expression. For example,

```
let age = 5;

let welcome = (age < 18) ?   () => alert('Baby') :  () => alert('Adult');
welcome();
 // Baby
```

CallBack Function

In JavaScript, you can also pass a function as an argument to a function. This function that is passed as an argument inside of another function is called a callback function.

For example,

```
// function
function greet(name, callback) {
   alert('Hi' + ' ' + name);
   callback();
}
```

```
// callback function
function callMe() {
    alert('I am callback function');
}

// passing function as an argument
greet('Sachin', callMe);
```

Run Code

Output

```
Hi Sachin
I am callback function
```

In the above program, there are two functions. While calling the greet() function, two arguments (a string value and a function) are passed.

The callMe() function is a callback function

Benefit of Callback Function

The benefit of using a callback function is that you can wait for the result of a previous function call and then execute another function call.

JavaScript Spread Operator

The spread operator ... is used to expand or spread an iterable or an array. For example,

```
const arrValue = ['My', 'name', 'is', 'Jack'];

console.log(arrValue);   // ["My", "name", "is", "Jack"]
console.log(...arrValue); // My name is Jack
```

In this case, the code:

```
console.log(...arrValue)
```

is equivalent to:

```
console.log('My', 'name', 'is', 'Jack');
```

Copy Array Using Spread Operator

You can also use the spread syntax ... to copy the items into a single array. For example,

```
const arr1 = ['one', 'two'];
const arr2 = [...arr1, 'three', 'four', 'five'];

console.log(arr2);
```

```
//  Output:
//  ["one", "two", "three", "four", "five"]
```

Clone Array Using Spread Operator

In JavaScript, objects are assigned by reference and not by values. For example,

```
let arr1 = [ 1, 2, 3];
let arr2 = arr1;

console.log(arr1); // [1, 2, 3]
console.log(arr2); // [1, 2, 3]

// append an item to the array
arr1.push(4);

console.log(arr1); // [1, 2, 3, 4]
console.log(arr2); // [1, 2, 3, 4]
```

Here, both variables arr1 and arr2 are referring to the same array. Hence the change in one variable results in the change in both variables.

However, if you want to copy arrays so that they do not refer to the same array, you can use the spread operator.

This way, the change in one array is not reflected in the other. For example,

```
let arr1 = [ 1, 2, 3];

// copy using spread syntax
let arr2 = [...arr1];

console.log(arr1); // [1, 2, 3]
console.log(arr2); // [1, 2, 3]

// append an item to the array
arr1.push(4);

console.log(arr1); // [1, 2, 3, 4]
console.log(arr2); // [1, 2, 3]
```

Spread Operator with Object

You can also use the spread operator with object literals. For example,

```
const obj1 = { x : 1, y : 2 };
const obj2 = { z : 3 };
```

```
// add members obj1 and obj2  to obj3
const obj3 = {...obj1, ...obj2};

console.log(obj3); // {x: 1, y: 2, z: 3}
```

Here, both obj1 and obj2 properties are added to obj3 using the spread operator.

Rest Parameter

When the spread operator is used as a parameter, it is known as the rest parameter.

You can also accept multiple arguments in a function call using the rest parameter. For example,

```
let func = function(...args) {
   console.log(args);
}

func(3); // [3]
func(4, 5, 6); // [4, 5, 6]
```

Here,

When a single argument is passed to the func() function, the rest parameter takes only one parameter.

When three arguments are passed, the rest parameter takes all three parameters.

Using the rest parameter will pass the arguments as array elements.

You can also pass multiple arguments to a function using the spread operator. For example,

```
function sum(x, y ,z) {
   console.log(x + y + z);
}

const num1 = [1, 3, 4, 5];

sum(...num1); // 8
```

If you pass multiple arguments using the spread operator, the function takes the required arguments and ignores the rest.

JavaScript Destructuring

The destructuring assignment introduced in ES6 makes it easy to assign array values and object properties to distinct variables. For example,

Before destructuring assignment :

```
// assigning object attributes to variables
const person = {
   name: 'Sara',
   age: 25,
   gender: 'female'
}

let name = person.name;
let age = person.age;
let gender = person.gender;

console.log(name); // Sara
console.log(age); // 25
console.log(gender); // female
```

<u>After destructuring assignment:</u>

```
// assigning object attributes to variables
const person = {
   name: 'Sara',
   age: 25,
   gender: 'female'
}

// destructuring assignment
let { name, age, gender } = person;

console.log(name); // Sara
console.log(age); // 25
console.log(gender); // female
```

The order of the name does not matter in object destructuring.

For example, you could write the above program as:

```
let { age, gender, name } = person;
console.log(name); // Sara
```

When destructuring objects, you should use the same name for the variable as the corresponding object key.

For example,

```
let {name1, age, gender} = person;
console.log(name1); // undefined
```

If you want to assign different variable names for the object key, you can use:

```
const person = {
   name: 'Sara',
   age: 25,
   gender: 'female'
}

// destructuring assignment
// using different variable names
let { name: name1, age: age1, gender:gender1 } = person;

console.log(name1); // Sara
console.log(age1); // 25
console.log(gender1); // female
```

<u>**Array Destructuring**</u>

You can also perform array destructuring in a similar way. For example,

```
const arrValue = ['one', 'two', 'three'];

// destructuring assignment in arrays
const [x, y, z] = arrValue;

console.log(x); // one
console.log(y); // two
console.log(z); // three
```

<u>Assign Default Values</u>

You can assign the default values for variables while using destructuring. For example,

```
let arrValue = [20];

// assigning default value 5 and 7
let [x = 5,  y = 7] = arrValue;
```

```
console.log(x); // 20
console.log(y); // 7
```

In the above program, arrValue has only one element. Hence,

the x variable will be 10

the y variable takes the default value 7

<u>In object destructuring, you can pass default values in a similar way.</u>

For example,

```
const person = {
    name: 'Jack',
}

// assign default value 26 to age if undefined
const { name, age = 26} = person;

console.log(name); // Jack
console.log(age); // 26
```

JavaScript Classes

JavaScript Class Syntax

Use the keyword class to create a class.

Always add a method named constructor():

Example

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
}
const myCar1 = new Car("Ford", 2014);
const myCar2 = new Car("Audi", 2019);
```

The example above uses the Car class to create two Car objects.

Class Methods

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

```
     }
age()
 {
     const date = new Date();
     return date.getFullYear() - this.year;
   }
  }


  const myCar = new Car("Ford", 2019);
  document.write("My car is " + myCar.age() + " years
old.");
```

Class Inheritance

Inheritance enables you to define a class that takes all the functionality from a parent class and allows you to add more.

Using class inheritance, a class can inherit all the methods and properties of another class.

Inheritance is a useful feature that allows code reusability.

To use class inheritance, you use the extends keyword. For example,

```
// parent class
class Person {
    constructor(name) {
        this.name = name;
    }

    greet() {
        console.log(`Hello ${this.name}`);
    }
}

// inheriting parent class
class Student extends Person {
}

let student1 = new Student('Jack');
student1.greet();
```

Output

```
Hello Jack
```

In the above example, the Student class inherits all the methods and properties of the Person class. Hence, the Student class will now have the name property and the greet() method.

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.

Export

Modules with functions or variables can be stored in any external file.

There are two types of exports: Named Exports and Default Exports.

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 individually:

person.js

```
export const name = "Sachin";
export const age = 20;
```

All at once at the bottom:

person.js

```
const name = "Sachin";
const age = 20;

export {name, age};
```

Default Exports

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

You can only have one default export in a file.

Example

msg.js

```
const message = () => {
const name = "Sachin";
const age = 25;
return name + ' is ' + age + 'years old.';
};

export default message;
```

Import

You can import modules into a file in two ways, based on if they are named exports or default exports.

Named exports are constructed using curly braces. Default exports are not.

Import from named exports

Import named exports from the file person.js:

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

module_use1.html

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Modules</h1>

<p id="demo"></p>

<script type="module">
import { name, age } from "./person.js";
```

```
let text = "My name is " + name + ", I am " + age + ".";

document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

<u>Import from default exports</u>

Import a default export from the file msg.js:

module_use2.html

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Modules</h1>

<p id="demo"></p>

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

```
document.getElementById("demo").innerHTML =
message();
</script>
</body>
</html>
```

JavaScript and JSON

JSON stands for Javascript Object Notation. JSON is a text-based data format that is used to store and transfer data. For example,

```
// JSON syntax
{
   "name": "John",
   "age": 22,
   "gender": "male",

}
```

In JSON, the data are in key/value pairs separated by a comma ,.

JSON was derived from JavaScript. So, the JSON syntax resembles JavaScript object literal syntax. However, the JSON format can be accessed and be created by other programming languages too.

<u>JSON Data</u>

JSON data consists of key/value pairs similar to JavaScript object properties. The key and values are written in double quotes separated by a colon :.

For example,

```
// JSON data
"name": "Sachin"
```

Note: JSON data requires double quotes for the key.

<u>JSON Object</u>

The JSON object is written inside curly braces { }. JSON objects can contain multiple key/value pairs. For example,

```
// JSON object
```

```
{ "name": "Sachin", "age": 22 }
```

JSON Array

JSON array is written inside square brackets [ ]. For example,

```
// JSON array
[ "apple", "mango", "banana"]

// JSON array containing objects
[
   { "name": "Sachin", "age": 22 },
   { "name": "Anil", "age": 20 }.
   { "name": "Mark", "age": 23 }
]
```

Accessing JSON Data

You can access JSON data using the dot notation. For example,

```
// JSON object
const data = {
    "name": "Sachin",
    "age": 22,
    "hobby": {
        "reading" : true,
        "gaming" : false,
        "sport" : "football"
    },
    "class" : ["JavaScript", "HTML", "CSS"]
}

// accessing JSON object

document.write("<br>"+data.name); // Sachin
document.write("<br>"+data.hobby.sport);
// football
document.write("<br>"+data.class[1]); // HTML
```

We use the . notation to access JSON data. Its syntax is: variableName.key

You can also use square bracket syntax [] to access JSON data. For example,

```
// JSON object
```

```
const data = {
   "name": " Sachin ",
   "age": 22
}

// accessing JSON object
document.write("<br>"+data["name"]); // Sachin
```

JavaScript Objects VS JSON

Though the syntax of JSON is similar to the JavaScript object, JSON is different from JavaScript objects.

JSON	JavaScript Object
The key in key/value pair should be in double quotes.	The key in key/value pair can be without double quotes.
JSON cannot contain functions.	JavaScript objects can contain functions.
JSON can be created and used by other programming languages.	JavaScript objects can only be used in JavaScript.

Use of JSON

JSON is the most commonly used format for transmitting data (data interchange) from a server to a client and vice-versa. JSON data are very easy to parse and use. It is fast to access and manipulate JSON data as they only contain texts.

JSON is language independent. You can create and use JSON in other programming languages too.