ES6 (ECMAScript)

ES6

ES6 or ECMAScript6 is a scripting language specification which is standardized by ECMAScript International. It is used by the applications to enable client-side scripting. This specification is affected by programming languages like Self, Perl, Python, Java, etc. This specification governs some languages such as JavaScript, ActionScript, and Jscript. **ECMAScript** is generally used for client-side scripting (React/Angular) and it is also used for writing server applications and services by using **NodeJS**.

ES6

In June 2015, there was a significant update to JavaScript, ushering in tons of new features including arrow functions, class destruction, and template strings, and more. Over the past years, these features continue to be updated to make your work with JS easier.

Overview of JavaScript Updates

JavaScript (JS) is a lightweight, object-oriented, interpreted programming language. In 1996, Netscape submitted JS to ECMA International to be standardized, which lead to a new version of the language called ECMAScript.

It took nearly 15 years for ECMAScript to see major changes, and since then, it has been updated regularly.

History

The ECMAScript specification is the standardized specification of scripting language, which is developed by Brendan Eich (He is an American technologist and the creator of JavaScript programming language) of Netscape (It is a name of brand which is associated with Netscape web browser's development).

Initially, the ECMAScript was named Mocha, later LiveScript, and finally, JavaScript. In December 1995, Sun Microsystems (an American company that sold the computers and its components, software, and IT services. It created Java, NFS, ZFS, SPARC, etc.) and Netscape announced the JavaScript during a press release.

During November 1996, Netscape announced a meeting of the ECMA International standard organization to enhance the standardization of JavaScript.

History

ECMA General Assembly adopted the first edition of ECMA-262 in June 1997. Since then, there are several editions of the language standard have published. The Name 'ECMAScript' was a settlement between the organizations which included the standardizing of the language, especially Netscape and Microsoft, whose disputes dominated the primary standard sessions. Brendan Eich commented that 'ECMAScript was always an unwanted trade name which sounds like a skin disease (eczema).’

Both JavaScript and Jscript aims to be compatible with the ECMAScript, and they also provide some of the additional features that are not described in ECMA specification.

ECMAScript 2

The first standardized version of ECMAScript was released in 1997. ECMAScript 2 followed a year later, bringing minor changes to modernize the language with ISO standards.

ECMAScript 3

ECMAScript 3 was released in 1999 and ushered in many new popular features, including expression, try/catch exception handling, and more. After ECMAScript 3, no changes were made to the official standard for many years.

ECMAScript 4

ECMAScript 4 was proposed as a significant upgrade in the mid-2000s. There was some controversy over these updates, and ES4 was scrapped.

ECMAScript 5

ECMAScript 5 (ES5) came along in 2009 with subtle changes to ES3 so that JavaScript could be supported in all browsers. The desire for a more robust update began around 2012 when there was a stronger push to abandon support for Internet Explorer.

ECMAScript 6

The next big update occurred in 2015 when ECMAScript 6 (ES6) or ECMAScript 2015 (ES2015) was officially released. ES6 features modernized JavaScript.

ECMAScript 7

The 7th edition was officially known as ECMAScript 2016, which was finalized in June 2016. The standard language includes features such as block scoping of functions and variables, destructing the patterns of variables, proper tail calls, **async/await** keywords for asynchronous programming, exponentiation operator ** for numbers.

ECMAScript 8

The 8th edition was officially known as ECMAScript 2017, which was finalized in June 2017. It includes the async/await constructions which work using promises (In CS future, promise, deferred, and delay refers to the constructs which are used to synchronize the execution of the program in some concurrent programming languages) and **generators**.

ECMAScript 2017 or the eight edition also includes the features of atomic and concurrency, syntactic integration with promises.

ECMAScript 9

The 9th edition was officially known as ECMAScript 2018, which was finalized in June 2018. It includes the new features like **rest/spread operators** for the variables (three dots: …identifier), **asynchronous iteration**, etc.

ECMAScript 10

The 10th edition was officially known as ECMAScript 2019, which was published in June 2019. It includes the addition of some new features like **Array.prototype.flatMap**, **Array.prototype.flat**, and changes to **Array.sort** and **Object.fromEntries**.

ES Next

It is nothing but a dynamic name that refers to the next version at the writing time. The harmony plans were too committed for the single version, which splits its features within the two groups: the first group had the highest priority and was to become the succeeding version after ES5. ECMAScript.next was the code name of that version, for avoiding the prematurely committing to a version number which already proved suspect with ES4. The second group had time until after ECMAScript.next.

JavaScript

Just install nodemon and run
nodemon your_file.js
on vs code terminal.

ES6 Introduction

- ECMAScript 2015 was the second major revision to JavaScript.
- ECMAScript 2015 is also known as ES6 and ECMAScript 6.
- ECMAScript 2015 or ES2015 is a significant update to the JavaScript programming language. It is the first major update to the language since ES5 which was standardized in 2009. Therefore, ES2015 is often called ES6.

ES6 features

- The let keyword
- The const keyword
- Arrow Functions
- For/of
- Map Objects
- Set Objects
- Classes
- Promises
- Symbol
- Default Parameters
- Function Rest Parameter
- String.includes()

ES6 features

The let keyword:

- In ES5, when you declare a variable using the var keyword, the scope of the variable is either global or local. If you declare a variable outside of a function, the scope of the variable is global. When you declare a variable inside a function, the scope of the variable is local.
- ES6 provides a new way of declaring a variable by using the let keyword. The let keyword is similar to the var keyword, except that these variables are blocked-scope.
  - Let x = 10;
- In JavaScript, blocks are denoted by curly braces {}

JavaScript Variables

JavaScript variables are loosely typed, that is to say, variables can hold values with any type of data. Variables are just named placeholders for values.

Declare JavaScript variables using var keyword

To declare a variable, you use the var keyword followed by the variable name as follows:

var message;

A variable name can be any valid identifier. The message variable is declared and hold a special value undefined.

After declaring a variable, you can assign the variable a string as follows:

message = "Hello";

To declare a variable and initialize it at the same time, you use the following syntax:

var variableName = value;

JavaScript Variables

For example, the following statement declares the message variable and assign it a value "Hello"

var message = "Hello";

var counter = 100;

You can declare two or more variables using one statement, each variable declaration is separated by a comma (,) as follows:

var message = "Hello", counter = 100;

As mentioned earlier, you can store a number in the message variable as the following example though it is not recommended.

message = 100;

JavaScript Variables

Undefined vs. undeclared variables

It’s important to distinguish between undefined and undeclared variables.

An undefined variable is a variable that has been declared. Because we have not assigned it a value, the variable used the undefined as its initial value.

In contrast, an undeclared variable is the variable that has not been declared.

See the following example:

```
var message;

console.log(message); // undefined

console.log(counter); // ReferenceError: counter is not defined
```

In this example, the message variable is declared but not initialized therefore its value is undefined whereas the counter variable has not been declared hence accessing it causes a ReferenceError.

JavaScript Variables

Global and local variables

In JavaScript, all variables exist within a scope that determines the lifetime of the variables and which part of the code can access them.

JavaScript mainly has global and function scopes. ES6 introduced a new scope called block scope.

If you declare a variable in a function, JavaScript adds the variable to the function scope. In case you declare a variable outside of a function, JavaScript adds it to the global scope.

In JavaScript, you define a function as follows:

```
function functionName() {
  // logic
}
```

and call the function using the following syntax:

```
functionName();
```

JavaScript Variables

The following example defines a function named say that has a local variable named message.

```
function say() {
  var message = "Hi";
  return message;
}
```

The message variable is a local variable. In other words, it only exists inside the function.

If you try to access the message outside the function as shown in the following example, you will get a ReferenceError because the message variable was not defined:

```
function say() {
    var message = 'Hi';
}
console.log(message); // ReferenceError
```

JavaScript Variables

Variable shadowing:

```
// global variable
var message = "Hello";
function say() {
    // local variable
    var message = 'Hi';
    console.log(message); // which message?
}
say();// Hi
console.log(message); // Hello
```

In this example, we have two variables that share the same name: message. The first message variable is a global variable whereas the second one is the local variable.

Inside the say() function, the global message variable is shadowed. It cannot be accessible inside the say() function but outside of the function. This is called variable shadowing.

JavaScript Variables

Accessing global variable inside the function

```
// global variable
var message = "Hello";
function say() {
    // local variable
    message = 'Hi';
    console.log(message); // which message?
}
say();// Hi
Console.log(message); // Hi
```

In this example, we define a global variable named message. In the say() function, we reference the global message variable by omitting the var keyword and change its value to a string of Hi.

Although it is possible to refer to a global variable inside a function, it is not recommended. This is because the global variables are very difficult to maintain and potentially cause much confusion.

JavaScript Variables

Non-strict mode

The following example defines a function and declares a variable message. However, the var keyword is not used.

```
function say() {
    message = 'Hi'; // what?
    console.log(message);
}
say(); // Hi
console.log(message); // Hi
```

When you execute the script, it outputs the Hi string twice in the output.

Because when we call the say() function, the JavaScript engine looks for the variable named message inside the scope of the function.

As a result, it could not find any variable declared with that name so it goes up to the next immediate scope which is the global scope in this case and asks whether or not the message variable has been declared.

Because the JavaScript engine couldn’t find any of global variable named message so it creates a new variable with that name and adds it to the global scope.

JavaScript Variables

strict mode

To avoid creating a global variable accidentally inside a function because of omitting the var keyword, you use the strict mode by adding the "use strict"; at the beginning of the JavaScript file (or the function) as follows:

```
"use strict";
function say() {
    message = 'Hi'; // ReferenceError
    console.log(message);
}
say(); // Hi
console.log(message); // Hi
```

From now on, you should always use the strict mode in your JavaScript code to eliminate some JavaScript silent errors and make your code run faster.

JavaScript Variables

JavaScript variable hoisting

When executing JavaScript code, the JavaScript engine goes through two phases:

- Parsing
- Execution

In the parsing phase, The JavaScript engine moves all variable declarations to the top of the file if the variables are global, or to the top of a function if the variables are declared in the function.

In the execution phase, the JavaScript engine assigns values to variables and execute the code.

Hoisting is a mechanism that the JavaScript engine moves all the variable declarations to the top of their scopes, either function or global scopes.

If you declare a variable with the var keyword, the variable is hoisted to the top of its enclosing scope, either global or function scope.

As a result, if you access a variable before declaring it, the variable evaluates to undefined.

See the following example:

```
console.log(message); // undefined

var message;
```

JavaScript Variables

The JavaScript engine moves the declaration of the message variable to the top, so the above code is equivalent to the following:

```
var message;

console.log(message); // undefined
```

If there were no hoisting, you would get a ReferenceError because you referenced to a variable that was not defined.

See another example:

```
console.log(counter);

var counter = 100;
```

The JavaScript engine moves only the declaration of the variables to the top. However, it keeps the initial assignment of the variable remains intact. As a result, the code above is equivalent to the following code:

```
var counter;

console.log(counter); // undefined

counter = 100;
```

JavaScript Variables

The hoisting uses redundant var declarations without any penalty:

```
var counter;
var counter;
counter = 1;
console.log(counter); // 1
```

ES6 features

The let keyword:

- JavaScript let and global object:
  - When you declare a global variable using the var keyword, you add that variable to the property list of the global object. In the case of the web browser, the global object is the window.

```
var a = 50;
console.log(globalThis.a); // 50
```

  - when you use the let keyword to declare a variable, that variable is not attached to the global object as a property

```
let b = 20;
console.log(globalThis.b); // undefined
```

ES6 features

JavaScript globalThis :

- JavaScript let and global object:
    - ES2020 introduced the **globalThis** object that provides a standard way to access the global object across environments
    - In the web browsers, the global object is **window** or **framesIn** the web browsers, the global object is window or frames
    - The Web Workers API, however, doesn’t have the **window** object because it has no browsing context. Therefore, the Web Workers API uses **self** as a global object.
    - Node.js, on the other hand, uses the **global** keyword to reference the global object.
    - If you write JavaScript code that needs to access the global object, you have to use different syntaxes like window, frames, self, or global.
    - To standardize this, ES2020 introduced the **globalThis** that is available across environments.

ES6 features

The let keyword:

- JavaScript let and callback function in a for loop:

```
for (var i = 0; i < 5; i++) {
    setTimeout(function () {
        console.log(i);
    }, 1000);
}

for (var i = 0; i < 5; i++) {
    setTimeout(() => console.log(i), 1000);
}
```

ES6 features

The let keyword:

- JavaScript let and callback function in a for loop:

JS function:

```
   myFunction = function(){
        return “Hello Friends”;
}
```

Arrow function:

```
 myFunction = () => {
        return “Hello Friends”;
}
```

Arrow Functions Return Value by Default:

```
myFunction = () => “Hello Friends”;
```

Arrow Function With Parameters:

```
myFunction = (val) => “Hello ”+ val;
```

Arrow Function Without Parentheses:

```
myFunction = val => “Hello ”+ val;
```

ES6 features

The let keyword:

- JavaScript let and callback function in a for loop:

    Callback function:

    "I will call back later!"

    A callback is a function passed as an argument to another function

    This technique allows a function to call another function.

    A callback function can run after another function has finished

ES6 features

The let keyword:

- JavaScript let and callback function in a for loop:

  Callback:

  A callback is a function passed as an argument to another function.

  Using a callback, you could call the calculator function (myCalculator) with a callback, and let the calculator function run the callback after the calculation is finished:

  ```
  function myDisplayer(some) {
    console.log(some);
  }
  function myCalculator(num1, num2, myCallback) {
    let sum = num1 + num2;
    myCallback(sum);
  }
  myCalculator(5, 5, myDisplayer);
  ```

  Note: Where callbacks really shine are in asynchronous functions, where one function has to wait for another function (like waiting for a file to load).

ES6 features

The let keyword:

- JavaScript let and callback function in a for loop:

```
for (var i = 0; i < 5; i++) {
    setTimeout(?, 1000);
    setTimeout(200, 1000);
    setTimeout(function () { console.log(i);}, 1000)
}

for (var i = 0; i < 5; i++) {
    setTimeout(() => console.log(i), 1000);
}
Output: 5,5,5,5,5
```

In this example, the variable **i** is a global variable. After the loop, its value is 5. When the callback functions are passed to the setTimeout() function executes, they reference the same variable i with the value 5.

ES6 features

The let keyword:

- JavaScript let and callback function in a for loop:

In ES6, the let keyword declares a new variable in each loop iteration. Therefore, you just need to replace the var keyword with the let keyword to fix the issue:

```
for (let i = 0; i < 5; i++) {
    setTimeout(() => console.log(i), 1000);
}
```

Output:

```
0,1,2,3,4
```

ES6 features

The let keyword:

- JavaScript let and Redeclaration:

The var keyword allows you to redeclare a variable without any issue:

```
var counter = 0;
var counter;
console.log(counter); // 0
```

However, redeclaring a variable using the let keyword will result in an error:

```
let counter = 0;
let counter;
console.log(counter);
Error: Uncaught SyntaxError: Identifier 'counter' has already been declared
```

ES6 features

The let keyword:

- Difference between var and let:

let	**var**
let is block-scoped.	var is function scoped.
let does not allow to redeclare variables.	var allows to redeclare variables.
Hoisting does not occur in let.	Hoisting occurs in var.

ES6 features

The Const keyword:

- Variables defined with const cannot be Redeclared.

- Variables defined with const cannot be Reassigned.

- Variables defined with const have Block Scope.

ES6 features

The Const keyword:

- Variables defined with const cannot be Redeclared. JavaScript const variables must be assigned a value when they are declared.
- Variables defined with const cannot be Reassigned.
- Variables defined with const have Block Scope.
- Ex:

```
const PI = 3.141592653589793;
PI = 3.14;      // This will give an error
PI = PI + 10;   // This will also give an error
```

ES6 features

The Const keyword: When to use ?

- *As a general rule, always declare a variables with const unless you know that the value will change.*

- Always use const when you declare:

- A new Array
- A new Object
- A new Function
- A new RegExp

ES6 features

The Const keyword: When to use ?

- The keyword const is a little misleading.
- It does not define a constant value. It defines a constant reference to a value.

- **Because of this you can NOT**:
- Reassign a constant value
- Reassign a constant array
- Reassign a constant object

- **But you CAN**:
- Change a constant array
- Change a constant object

ES6 features

The Const keyword:

Constant Arrays:

```
// You can create a constant array:
const cars = ["Saab", "Volvo", "BMW"];

// You can change an element:
cars[0] = "Toyota";

// You can add an element:
cars.push("Audi");
```

But you can NOT reassign the array:

```
cars = ["Toyota", "Volvo", "Audi"];    // ERROR
```

ES6 features

The Const keyword:

Constant Objects:

```
// You can create a const object:
const car = {type:"Fiat", model:"500", color:"white"};

// You can change a property:
car.color = "red";

// You can add a property:
car.owner = "Johnson";
```

But you can NOT reassign the object:

```
car = {type:"Volvo", model:"EX60", color:"red"};    // ERROR
```

ES6 operators

- The operator can be defined as a symbol that tells the system to implement a particular operation. In JavaScript, there is a rich set of operators, and by using specific operators, you can perform any particular task.
- The operators are used in the expressions for evaluating different operands.
- An expression is a kind of statement that returns a value. The expression includes:
- **Operators:** It is responsible for performing some operations on operands
- **Operands:** It represents the data.
- **For example:** Suppose an expression like **x*y.** In this expression, **x** and **y** are the **operands,** and the asterisk **(*)** symbol is the multiplication operator.

ES6 operators

Types of Operators

Operators in JavaScript can be classified as:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Bitwise Operators
- Type Operators
- Miscellaneous Operators

ES6 operators

Arithmetic Operators

- Arithmetic operators are the basic mathematical operators that are available in JavaScript ES6. These operators are responsible for performing all mathematical operations such as addition, subtraction, etc. in JavaScript.

Operators	Functions
+ (Addition)	It returns the sum of the value of operands
- (Subtraction)	It returns the difference between the value of operands
*** (Multiplication)**	It returns the product of operands values.
/ (Division)	It is used to perform division, and it returns quotient.
% (Modulus)	It also performs division and returns the remainder.
++ (Increment)	It increments the value of a variable by one.
- (Decrement)	It decrements the value of a variable by one.

ES6 operators

Arithmetic Operators

- Example:

```
var x = 30;

var y = 20 ;

console.log("Addition: " + (x + y) );

console.log("Subtraction: " + (x - y) );

console.log("Multiplication: " + (x * y) );

console.log("The Division will give you the quotient: " + (x / y) );

console.log("Modulus will give you the Remainder: " + (x % y) );

// pre-increment

console.log("Value of x after pre-increment: "  + (++x) );

// post-increment

console.log("Value of x after post-increment: " + (x++) );

// pre-decrement

console.log("Value of y after pre-decrement: "  + (--y) );

// post-decrement

console.log("Value of y after post-decrement: " + (y--) );
```

ES6 operators

Relational Operators

Relational operators are used for comparing the two values and return either true or false based on the expression. These operators are sometimes called Comparison Operators.

Operator	Function
> (Greater than)	It returns true if the left operand is greater than right operand else it returns false.
< (Less than)	It returns true if the left operand is smaller than right operand else it returns false.
>= (Greater than or equal to)	It returns true if the left operand is greater than or equal to right operand else it returns false.
<= (Less than or equal to)	It returns true if the left operand is smaller than or equal to right operand else it returns false.
== (Equality)	It returns true if the value of both operands is the same else it returns false.
!= (Not Equal to)	It returns true if the value of operands is not the same else it returns false.

ES6 operators

Relational Operators

Example:

```
var x = 20;
var y = 15;
console.log("Value of x: " + x);
console.log("Value of y: " + y);
var result = x > y;
console.log("x is greater than y: " + result);
result = x < y;
console.log("x is smaller than y: " + result);
result = x >= y;
console.log("x is greater than or equal to  y: " + result);
result = x <= y;
console.log("x is smaller than or equal to y: " + result);
result = x == y;
console.log("x is equal to y: " + result);
result = x != y;
console.log("x not equal to  y: " + result);
```

ES6 operators

Logical Operators

Logical operators are generally used for combining two or more relational statements. They also return Boolean values.

Operators	Description
&& (Logical AND)	This operator returns true if all relational statements that are combined with && are true, else it returns false.
|| (Logical OR)	This operator returns true if at least one of the relational statements that are combined with || are true, else it returns false.
! (Logical NOT)	It returns the inverse of the statement's result.

ES6 operators

Logical Operators

Example:

```
var x = 30;
var y = 80;
console.log("Value of x = " + x );
console.log("Value of y = " + y );
var result = ((x < 40) && (y <= 90));
console.log("(x < 40) && (y <= 90): ", result);
var result = ((x == 50) || (y > 80));
console.log("(x == 50) || (y > 80): ", result);
var result = !((x > 20) && (y >= 80));
console.log("!((x > 20) && (y >= 80)): ", result);
```

ES6 operators

Assignment Operators

Assignment operators are used for assigning a value to the variable. The operand on the left side of the assignment operator is a variable, and the operand on the right side of the assignment operator is a value.

The right-side value must be of the same data-type of the left-side variable; otherwise, the compiler will raise an error.

Operators	Functions
= (Simple Assignment)	It simply assigns the value of the right operand to the left operand
+= (Add and Assignment)	This operator adds the value of the right operand to the value of the left operand and assigns the result to the left operand.
-= (Subtract and Assignment)	This operator subtracts the value of the right operand from the value of the left operand and assigns the result to the left operand.
***= (Multiply and Assignment)**	This operator multiplies the value of the right operand to the value of the left operand and assigns the result to the left operand.
/= (Divide and Assignment)	This operator divides the value of the right operand to the value of the left operand and assigns the result to the left operand.

ES6 operators

Assignment Operators

Example:

```
var x = 20;
var y = 40;
x = y;
console.log("After assignment the value of x is:  " + x);
x += y;
console.log("x+=y: " + x);
x -= y;
console.log("x-=y: " + x);
x *= y;
console.log("x*=y: " + x);
x /= y;
console.log("x/=y: " + x);
x %= y;
console.log("x%=y: " + x);
```

ES6 operators

Bitwise Operators

- Bitwise operators are used for performing the bitwise operations on binary numerals or bit patterns that involve the manipulation of individual bits. Bitwise operators perform the operation on the binary representation of arguments
- Generally, bitwise operators are less used and relevant for the applications and hyper-performance programs.

Operator	Description
Bitwise AND (&)	It compares every bit of the first operand to the corresponding bit of the second operand. If both of the bits are 1, then the result bit will set to 1, else it will set to 0.
Bitwise OR (|)	It compares every bit of the first operand to the corresponding bit of the second operand. If both of the bits are 0, then the result bit will set to 0, else it will set to 1.
Bitwise XOR (^)	It takes two operands and does XOR on each bit of both operands. It returns 1 if both of the two bits are different and returns 0 in any other case.
Bitwise NOT (~)	It flips the bits of its operand, i.e., 0 becomes 1 and 1 becomes 0.
Left shift (<<)	It shifts the value of the left operand to the left by the number of bits specified by the right operand.
Sign-propagating Right shift (>>)	It shifts the value of the left operand to the right by the number of bits specified by the right operand. This is sign-propagating because the bits that we are adding from the left depends upon the sign of the number (0 represents positive, and 1 represents negative).
Zero-fill right shift	It accepts two operands. The first operand specifies the number, and the second operator determines the number of bits to shift. Every bit gets shifted towards the right, and the overflowing bits will be discarded. Because the **0-bit** is added from the left, that's why it is a **zero-fill right shift.**

ES6 operators

Bitwise Operators

Example:

```
var x = 70; /* 70 = 0100 0110 */
var y = 80; /* 80 = 0101 0000 */
var res = 0;
console.log("Value of 70 in binary 0100 0110" );
console.log("Value of 80 in binary 0101 0000" );
res = x & y;      /* 64 = 0100 0000 */
console.log("Value of x & y = %d\n", res );
res = x | y;      /* 86 = 0101 0110 */
console.log("Value of x | y = %d\n", res );
res = x ^ y;      /* 22 = 0001 0110 */
console.log("Value of x ^ y = %d\n", res );
res = ~x;         /*-71 = -0100 0111 */
console.log("Value of ~ x = %d\n", res );
res = x << 2;     /* 280 = 1000 11000 */
console.log("Value of x << 2 = %d\n", res );
res = x >> 2;     /* 17 = 0001 0001 */
console.log("Value of x >> 2 = %d\n", res );
```

ES6 operators

Miscellaneous Operators

- These are the operators that perform different operations in different circumstances.

Operators	Description
+ (Concatenation Operator)	It applies to strings and appends the second string to first.
- (Negation Operator)	It changes the sign of the value.
? (Conditional Operator)	It is used for representing the conditional expression. It is also called a **ternary operator.**

ES6 features

Objects: (Real Life Objects, Properties, and Methods)

- In real life, a car is an object.
- A car has properties like weight and color, and methods like start and stop:
- All cars have the same properties, but the property values differ from car to car.
- All cars have the same methods, but the methods are performed at different times.

Object	Properties	Methods

	car.name = Fiat
	
	car.model = 500
	
	car.weight = 850kg
	
	car.color = white
		car.start()
		
		car.drive()
		
		car.brake()
		
		car.stop()

ES6 features

JavaScript Objects:

```
const person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
const person = {
        firstName: "John",
        lastName: "Doe",
        age: 50,
        eyeColor: "blue‟,
        fullName : function() {
                return this.firstName + " " + this.lastName;
        }
};
```

ES6 features

JavaScript Objects:

The this Keyword:

- In a function definition, this refers to the "owner" of the function.
- In other words, **this.firstName** means the firstName property of this object.

Accessing Object Methods:

objectName.methodName(); example: name = person.fullName();

Do Not Declare Strings, Numbers, and Booleans as Objects!

When a JavaScript variable is declared with the keyword "new", the variable is created as an object:

```
x = new String();        // Declares x as a String object
y = new Number();        // Declares y as a Number object
z = new Boolean();       // Declares z as a Boolean object
```

Avoid String, Number, and Boolean objects. They complicate your code and slow down execution speed.

ES6 features

JavaScript Built-in native Objects:

Number:

In the place of number, if you provide any non-number argument, then the argument cannot be converted into a number, it returns NaN (Not-a-Number).

var val1 = new Number(number);

Methods:

toExponential() –

toFixed()

toLocaleString()

toPrecision()

toString

valueOf()

ES6 features

JavaScript Built-in native Objects:

String:

The String object lets you work with a series of characters; it wraps Javascript's string primitive data type with a number of helper methods.

As JavaScript automatically converts between string primitives and String objects, you can call any of the helper methods of the String object on a string primitive

var val1 = new String(string);

Methods:

chartAt() –

indexOf() -

lastIndexOf() –

localeCompare()-

match()-

replace()-

search()-

ES6 features

JavaScript Built-in native Objects:

Arrays:

The Array object lets you store multiple values in a single variable. It stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

var fruits = new Array("apple", "orange", "mango");

Var number = [1,2,3,4,5,1,2];

Methods:

concat() –

indexOf(1) -

lastIndexOf(1) –

//localeCompare()-

pop()-

push()-

reverse()-

ES6 features

JavaScript Built-in native Objects:

Date: The Date object is a datatype built into the JavaScript language. Date objects are created with the new Date( ) as shown below.

new Date( )

new Date(milliseconds)

new Date(datestring)

new Date(year,month,date[,hour,minute,second,millisecond])

Methods:

getDate()-

getDay()-

getHours()-

getMilliseconds()-

getMinutes()-

Static Methods : parse() and UTC()

ES6 features

JavaScript Built-in native Objects:

Boolean:

The Boolean object represents two values, either "true" or "false". If value parameter is omitted or is 0, -0, null, false, NaN, undefined, or the empty string (""), the object has an initial value of false.

var val1 = new Boolean(value);

Methods:

toSource() –

toString() -

valueOf() -

Default Parameter

```
function say(message='Hi') {
    console.log(message);
}
say(); // 'Hi'
say('Hello') // 'Hello’
```

The default value of the message paramater in the say() function is 'Hi'.

In JavaScript, default function parameters allow you to initialize named parameters with default values if no values or undefined are passed into the function.

Arguments vs. Parameters

- Sometimes, you can use the term argument and parameter interchangeably. However, by definition, parameters are what you specify in the function declaration whereas the arguments are what you pass to the function.

```
function add(x, y) {
   return x + y;
}
add(100,200);
```

In this example, the x and y are the parameters of the add() function, and the values passed to the add() function 100 and 200 are the arguments.

Default Parameter

Setting JavaScript default parameters for a function
In JavaScript, a parameter has a default value of undefined. It means that if you don’t pass the arguments into the function, its parameters will have the default values of undefined.

```
function say(message) {
    console.log(message);
}
say(); // undefined
```

The say() function takes the message parameter. Because we didn’t pass any argument into the say() function, the value of the message parameter is undefined.

Suppose that you want to give the message parameter a default value 10.
A typical way for achieving this is to test parameter value and assign a default value if it is undefined using a ternary operator:

```
function say(message) {
    message = typeof message !== 'undefined' ? message : 'Hi';
    console.log(message);
}
say(); // ‘Hi’
```

In this example, we didn’t pass any value into the say() function. Therefore, the default value of the message argument is undefined. Inside the function, we reassigned the message variable the Hi string.

Default Parameter

ES6 provides you with an easier way to set the default values for the function parameters like this:

```
function fn(param1=default1, param2=default2,..) {
}
```

In the syntax above, you use the assignment operator (=) and the default value after the parameter name to set a default value for that parameter.

```
function say(message='Hi') {
    console.log(message);
}

say(); // 'Hi'
say(undefined); // 'Hi'
say('Hello'); // 'Hello’
```

How it works.

In the first function call, we didn’t pass any argument into the say() function, therefore message parameter took the default value 'Hi'.
In the second function call, we passed the undefined into the say() function, hence the message parameter also took the default value 'Hi'.
In the third function call, we passed the 'Hello' string into the say() function, therefore message parameter took the string 'Hello' as the default value.

Default Parameter

More JavaScript default parameter examples:

1) Passing undefined arguments:

The following createDiv() function creates a new <div> element in the document with a specific height, width, and border-style:

```
function createDiv(height = '100px', width = '100px', border = 'solid 1px red') {
    let div = document.createElement('div');
    div.style.height = height;
    div.style.width = width;
    div.style.border = border;
    document.body.appendChild(div);
    return div;
}
```

The following doesn’t pass any arguments to the function so the createDiv() function uses the default values for the parameters.

```
createDiv();
```

Suppose you want to use the default values for the height and width parameters and specific border style. In this case, you need to pass undefined values to the first two parameters as follows:

```
createDiv(undefined,undefined,'solid 5px blue’);
```

Default Parameter

2) Evaluating default parameters:

JavaScript engine evaluates the default arguments at the time you call the function.

```
function put(toy, toyBox = []) {
   toyBox.push(toy);
   return toyBox;
}

console.log(put('Toy Car'));
// -> ['Toy Car']
console.log(put('Teddy Bear'));
// -> ['Teddy Bear'], not ['Toy Car','Teddy Bear’]
```

The parameter can take a default value which is a result of a function.

```
function date(d = today()) {
   console.log(d);
}
function today() {
   return (new Date()).toLocaleDateString("en-US");
}
date();
```

Default Parameter

2) Evaluating default parameters:

The date() function takes one parameter whose default value is the returned value of the today() function. The today() function returns today’s date in a specified string format.

When we declared the date() function, the today() function has not yet evaluated until we called the date() function.

We can use this feature to make arguments are mandatory. If the caller doesn’t pass any argument, we throw an error as follows:

```
function requiredArg() {
  throw new Error('The argument is required');
}
function add(x = requiredArg(), y = requiredArg()){
  return x + y;
}

add(10); // error
add(10,20); // OK
```

Default Parameter

3) Using other parameters in default values

You can assign a parameter a default value that references to other default parameters as shown in the following example:

```
function add(x = 1, y = x, z = x + y) {
   return x + y + z;
}

console.log(add()); // 4
```

In the add() function:
The default value of the y is set to x parameter.
The default value of the z is the sum of x and y
The add() function returns the sum of x, y, and z.
The parameter list seems to have its own scope. If you reference the parameter that has not been initialized yet, you will get an error. For example:

```
function subtract( x = y, y = 1 ) {
   return x - y;
}
subtract(10);
```

Error message:

```
Uncaught ReferenceError: Cannot access 'y' before initialization
```

Default Parameter

Using functions
You can use a return value of a function as a default value for a parameter.

```
let taxRate = () => 0.1;
let getPrice = function( price, tax = price * taxRate() ) {
    return price + tax;
}

let fullPrice = getPrice(100);
console.log(fullPrice); // 110
```

In the getPrice() function, we called the taxRate() function to get the tax rate and used this tax rate to calculate the tax amount from the price.

The arguments object

The value of the arguments object inside the function is the number of actual arguments that you pass to the function. For example:

```
function add(x, y = 1, z = 2) {
    console.log( arguments.length );
    return x + y + z;
}

add(10); // 1
add(10, 20); // 2
add(10, 20, 30); // 3
```

ES6 Spread Operator

ES6 introduced a new operator referred to as a spread operator, which consists of three dots (...). It allows an iterable to expand in places where more than zero arguments are expected. It gives us the privilege to obtain the parameters from an array.

Spread operator syntax is similar to the rest parameter, but it is entirely opposite of it. Let's understand the syntax of the spread operator.

```
var variablename1 = [...value];
```

The three dots (...) in the above syntax are the spread operator, which targets the entire values in the particular variable.

ES6 Spread Operator

Spread Operator and Array Manipulation

Here, we are going to see how we can manipulate an array by using the spread operator.

Constructing array literal

When we construct an array using the literal form, the spread operator allows us to insert another array within an initialized array.

```
let colors = ['Red', 'Yellow'];
let newColors = [...colors, 'Violet', 'Orange', 'Green'];
console.log(newColors);
```

Concatenating arrays

Spread operator can also be used to concatenate two or more arrays.

```
let colors = ['Red', 'Yellow'];
let newColors = [...colors, 'Violet', 'Orange', 'Green'];
console.log(newColors);
```

Copying an array - copy the instance of an array by using the spread operator.

```
let colors = ['Red', 'Yellow'];

let newColors = [...colors];

console.log(newColors);
```

ES6 Spread Operator

Spread Operator and Array Manipulation

If we copy the array elements without using the spread operator, then inserting a new element to the copied array will affect the original array.
But if we are copying the array by using the spread operator, then inserting an element in the copied array will not affect the original array.

Without using spread operator

```
let colors = ['Red', 'Yellow'];
let newColors = colors;
newColors.push('Green');
console.log(newColors);
console.log(colors);
```

Using spread operator

```
let colors = ['Red', 'Yellow'];
let newColors = [...colors];
newColors.push('Green');
console.log(newColors);
console.log(colors);
```

Spread operator and Strings

```
let str = ['A', ...'EIO', 'U'];
console.log(str);
```

It spreads out each specific character of the **'EIO'** string into individual characters.

ES6 Spread Operator

JavaScript spread operator and apply() method

```
function compare(a, b) {
   return a - b;
}
```

In ES5, to pass an array of two numbers to the compare() function, you often use the apply() method as follows:

```
var result = compare.apply(null, [1, 2]);
console.log(result); // -1
```

However, by using the spread operator, you can pass an array of two numbers to the compare() function:

```
let result = compare(...[1, 2]);
console.log(result); // -1
```

The spread operator spreads out the elements of the array so a = 1 and b = 2 in this case.

ES6 Spread Operator

A better way to use the Array's push() method example

Sometimes, a function may accept an indefinite number of arguments. Filling arguments from an array is not convenient.

For example, the push() method of an array object allows you to add one or more elements to an array. If you want to pass an array to the push() method, you need to use apply() method as follows:

```
var rivers = ['Nile', 'Ganges', 'Yangte'];
var moreRivers = ['Danube', 'Amazon'];

Array.prototype.push.apply(rivers, moreRivers);
console.log(rivers);
```

This solution looks verbose.

The following example uses the spread operator to improve the readability of the code:

```
rivers.push(...moreRivers);
```

using the spread operator is much cleaner.

ES6 Spread Operator

With Object

```
//cloning
const circle = {
    radius: 10,
    style: {
        color: 'blue'
    }
};

const clonedCircle = {
    ...circle
};


clonedCircle.style = 'red';

console.log(clonedCircle);
```

ES6 Spread Operator

Merging Object

```
const circle = {
  radius: 10
};

const style = {
  backgroundColor: 'red'
};

const solidCircle = {
  ...circle,
  ...style
};

console.log(solidCircle);
```

ES6 Rest Parameter

The rest parameter is introduced in ECMAScript 2015 or ES6, which improves the ability to handle parameters. The rest parameter allows us to represent an indefinite number of arguments as an array. By using the rest parameter, a function can be called with any number of arguments.

Before ES6, the **arguments** object of the function was used. The **arguments** object is not an instance of the Array type. Therefore, we can't use the **filter()** method directly.

The rest parameter is prefixed with three dots (...). Although the syntax of the rest parameter is similar to the **spread operator**, it is entirely opposite from the spread operator.

The rest parameter has to be the last argument because it is used to collect all of the remaining elements into an array.

```
function show(...args) {
  let sum = 0;
  for (let i of args) {
      sum += i;
  }
  console.log("Sum = "+sum);
}
show(10, 20, 30);
```

All the arguments that we have passed in the function will map to the parameter list. As stated above, the rest parameter (...) should always be at last in the list of arguments. If we place it anywhere else, it will cause an error.

Rest Parameters and Destructuring

Destructuring means to break down a complex structure into simpler parts. We can define an array as the rest parameter. The passed-in arguments will be broken down into the array. Rest parameter supports array destructuring only.

By using the rest parameter, we can put all the remaining elements of an array in a new array.

ES6 Rest Parameter

Rest Parameters and Destructuring

Destructuring means to break down a complex structure into simpler parts. We can define an array as the rest parameter. The passed-in arguments will be broken down into the array. Rest parameter supports array destructuring only.

By using the rest parameter, we can put all the remaining elements of an array in a new array.

```
var colors = ["Violet", "Indigo", "Blue", "Green", "Yellow", "Orange", "Red"];
// destructuring assignment
var [a,b,...args] = colors;
console.log(a);
console.log(b);
console.log(args);
```

Rest Parameter in a dynamic function

JavaScript allows us to create dynamic functions by using the **function** constructor. We can use the rest parameter within a dynamic function.

```
let num = new Function('...args','return args');

console.log(num(10, 20, 30));
```

ES6 Rest Parameter

Rest Parameters examples

Destructuring means to break down a complex structure into simpler parts. We can define an array as the rest parameter. The passed-in arguments will be broken down into the array. Rest parameter supports array destructuring only.

By using the rest parameter, we can put all the remaining elements of an array in a new array.

```
function sum(...args) {
  let total = 0;
  for (const a of args) {
    total += a;
  }
  return total;
}

sum(1, 2, 3); //6
```

Assuming that the caller of the sum() function may pass arguments with various kind of data types such as number, string, and boolean, and you want to calculate the total of numbers only:

```
function sum(...args) {
        return args
        .filter(function (e) {
        return typeof e === "number";
        })
        .reduce(function (prev, curr) {
        return prev + curr;
        });
}
```

ES6 Rest Parameter

Rest Parameters examples

The following script uses the new sum() function to sum only numeric arguments:

```
let result = sum(10,'Hi',null,undefined,20);
console.log(result);//30
```

Note that without the rest parameters, you have to use the arguments object of the function.

However, the arguments object itself is not an instance of the Array type, therefore, you cannot use the filter() method directly. In ES5, you have to use Array.prototype.filter.call() as follows:

```
function sum() {
        return Array.prototype.filter
                .call(arguments, function (e) {
                        return typeof e === "number";
                })
                .reduce(function (prev, curr) {
                        return prev + curr;
                });
}
```

As you see, the rest parameter makes the code more elegant. Suppose you need to filter the arguments based on a specific type such as numbers, strings, boolean, and null. The following function helps you to do it:

```
function filterBy(type, ...args) {
        return args.filter(function (e) {
                return typeof e === type;
        });
}
```

ES6 Rest Parameter

Rest Parameters and arrow function:

An arrow function does not have the arguments object. Therefore, if you want to pass a number of arguments to the arrow function, you must use the rest parameters.

```
const combine = (...args) => {
            return args.reduce(function (prev, curr) {
                        return prev + " " + curr;
            });
};
let message = combine("JavaScript", "Rest", "Parameters"); // =>
console.log(message); // JavaScript Rest Parameters
```

The combine() function is an arrow that takes an indefinite number of arguments and concatenates these arguments.

JavaScript rest parameter in a dynamic function

JavaScript allows you to create dynamic functions through the Function constructor. And it is possible to use the rest parameter in a dynamic function. Here is an example:

```
var showNumbers = new Function('...numbers', 'console.log(numbers)');
showNumbers(1, 2, 3); //[1, 2, 3]
```

ES6 Rest Parameter

Difference between Rest Parameter and arguments object

The rest parameter and arguments object are different from each other. Let's see the difference between the rest parameter and the arguments object:

The arguments object is an array-like (but not array), while the rest parameters are array instances. The arguments object does not include methods such as sort, map, forEach, or pop, but these methods can be directly used in rest parameters.

Here are the main differences:

- The spread operator unpacks elements.
- The rest parameter packs elements into an array.
- The rest parameters must be the last arguments of a function. However, the spread operator can be used anywhere.

Switch Statement

The switch statement is used to perform different actions based on different conditions.

```
switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
    break;
  case 1:
    day = "Monday";
    break;
  case 2:
     day = "Tuesday";
    break;
  case 3:
    day = "Wednesday";
    break;
  case 4:
    day = "Thursday";
    break;
  case 5:
    day = "Friday";
    break;
  case 6:
    day = "Saturday";
  default:
    day= "Sunday";
}
```

This is how it works:

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.
- If there is no match, the default code block is executed.

Switch Statement

Common Code Blocks

```
switch (new Date().getDay()) {
  case 4:
  case 5:
    text = "Soon it is Weekend";
    break;
  case 0:
  case 6:
    text = "It is Weekend";
    break;
  default:
    text = "Looking forward to the Weekend";
}
```

Switching Details

If multiple cases matches a case value, the **first** case is selected.

If no matching cases are found, the program continues to the **default** label.

If no default label is found, the program continues to the statement(s) **after the switch**.

Switch Statement

Strict Comparison

Switch cases use **strict** comparison (===).

The values must be of the same type to match.

A strict comparison can only be true if the operands are of the same type.

In this example there will be no match for x:

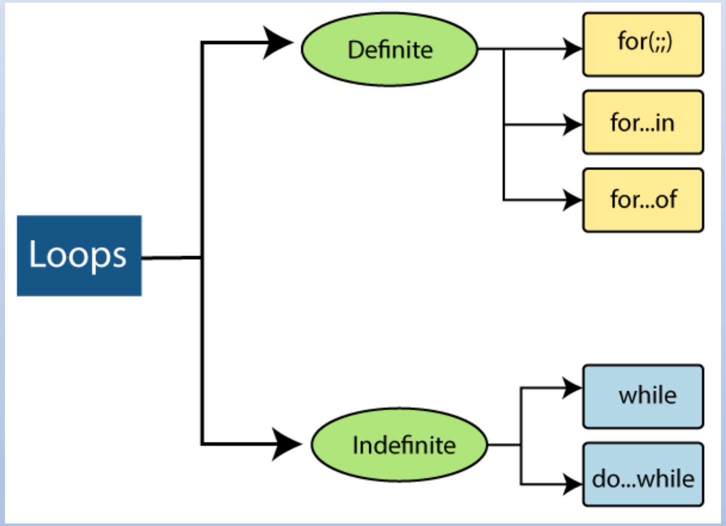
```
let x = "0";
switch (x) {
  case 0:
    text = "Off";
    break;
  case 1:
    text = "On";
    break;
  default:
    text = "No value found";
}
```

ES6 Loops

Looping statements in the programming languages help to execute the set of instructions/functions repeatedly while a condition evaluates to true. Loops are an ideal way to perform the execution of certain conditions repeatedly. In a loop, the repetition is termed as **iteration**

Definite Loops - A definite loop has a definite/fixed number of iterations.

Definite Loop	Description
for(; ;) Loop	It executes the block of code for a definite number of times.
for…in Loop	It iterates through the properties of the object.
for…of loop	Unlike object literals, it iterates the iterables (arrays, string, etc.).



The **for (; ;) loop** is used to iterate a part of the program multiple times. If you have a fixed number of iterations,

then it is always recommended to use 'for' loop.

```
"use strict"
for(let temp, a = 0, b = 1; b<40; temp = a, a = b, b = a + temp)
console.log(b);
//infinite loops
for(;;)
{
  console.log("infinitive loop");  // It will print infinite times
}
```

ES6 Loops

The for…in loop

The **for…in loop** is similar to **for loop,** which iterates through the properties of an object, i.e., when you require to visit the properties or keys of the object, then you can use **for…in loop.** It is a better choice when you are working with objects or dictionaries where the order of index is not essential.

```
function Mobile(model_no){
this.Model = model_no;
this.Color = 'White';
this.RAM = '4GB';
}
var Samsung = new Mobile("Galaxy");
for(var props in Samsung)
{
console.log(props+ " : " +Samsung[props]);
}
```

The for…of loop

Unlike the object literals, this loop is used to iterate the iterables (arrays, string, etc.).

```
var fruits = ['Apple', 'Banana', 'Mango', 'Orange'];
for(let value of fruits)
{
  console.log(value);
}
```

ES6 Loops

Indefinite Loops

An indefinite loop has infinite iterations. It is used when the number of iterations within the loop is intermediate or unknown.

Indefinite Loops	Description
while Loop	It executes the instructions each time till the defined condition evaluates to true.
do…while Loop	It is similar to the while loop, but the key difference is that the do…while loop executes the loop at once irrespective of the terminator condition.

The while loop

A while loop is a control flow statement that allows the repeated execution of code based on the given Boolean condition. It consists of a block of code and an expression/condition. The while loop checks the expression/condition before the execution of block; that's why this control structure is often known as a **pre-test loop**.

```
var y = 0;
while (y < 4) {
   console.log(y);
   y++;
}
```

The do…while loop

It is a control flow statement that executes a block of code at least once, and then it will depend upon the condition whether or not the loop executes the block repeatedly.

The do…while loop checks the condition after the execution of the block, that's why this control structure is also known as the **post-test loop.** It is also possible that the condition always evaluates to true, which will create an **infinite loop.**

ES6 Loops

The do…while loop

It is a control flow statement that executes a block of code at least once, and then it will depend upon the condition whether or not the loop executes the block repeatedly.

The do…while loop checks the condition after the execution of the block, that's why this control structure is also known as the **post-test loop.** It is also possible that the condition always evaluates to true, which will create an **infinite loop.**

```
var count = 6, fact = 1;

do {

  fact = fact * count--;

} while (count > 0);


console.log(fact);
```

The key difference between the these two that **while loop** is entered only if the condition passed to it will evaluate to true. But the **do…while loop** executes the statement once, it happens because of the starting iteration of the **do…while loop** does not consider as the Boolean expression. Then for the further iterations, the while will check the condition and takes the control out from the loop.

ES6 Loops

The Loop control statements

The loop control statements are used for interrupting or control the flow of the execution. These statements change the execution from its normal sequence. JavaScript provides you the full control to handle the loops and switch statements.

There may be some circumstances when you require to come out from the loop without reaching its bottom. There may also be some situations when you need to skip the part of the code and start the further iterations of the loop. So, for handling such situations in JavaScript, we have **a break** and **continue** statements.

Loop control statements	Description
The break statement	The break statement is used to bring the control of the program out from the loop.
The continue statement	It skips the subsequent statements of the current iteration and brings control of the program to the beginning of the loop.

The break statement

It is used to take control of the program out from the loop. You can use the break statements within the loops or in the switch statements. Using a break statement within the loop causes the program to exit from the loop.

```
var n = 1;
while(n<=7)
{
  console.log("n="+n);
  if(n==4)
  {
    break;
  }
  n++;
}
```

ES6 Loops

The continue statement

Unlike the break statement, the continue statement does not exit from the loop. It terminates the current iteration of the loop and starts the next iteration.

```
var n = 0;
while(n<=5)
{
  n++;
  if(n==3)
  {
     continue;
  }
  console.log("n = "+n);
}
```

Use of Labels for controlling the flow

Label	Description
Label with the break statement	It is used to exit from the loop or from the switch statement without using a label reference, but with label reference, it is used to jump out from any code block.
Label with continue statement	It is used to skip one iteration of the loop with or without using the label reference.

A **label** is nothing but an identifier followed by a **colon (:),** and it applied on a block of code or a statement. You can use the label with a **break** and **continue** to control the flow.

You cannot use the line breaks in between the break and continue statement and its label name. Also, there should not be any statement in between the label name and an **associated loop.**

```
var x, y;
loop1:       //The first for statement is labeled as "loop1."
for (x = 1; x < 4; x++) {
    loop2:   //The second for statement is labelled as "loop2"
  for (y = 1; y < 4; y++) {
    if (x === 2 && y === 2) {   break loop1;   }
    console.log('x = ' + x + ', y = ' + y);
  } }
```

ES6 Arrays

ES6 Arrays

Array in JavaScript is an object which is used to represent a collection of similar type of elements. It allows you to store more than one value or a group of values in a single variable name. Arrays are used for storing the collections of values in chronological order. An array is the collection of homogeneous elements, or we can say that array is the collection of values of same data-type.
We can store any valid values such as objects, numbers, strings, functions, and also other arrays, which make it possible to create complex data structures like an array of arrays or an array of objects.

```
var array_name = new Array();  // By using the new keyword
var array_name = [value1, value2,....valueN];  //By using Array literals
var array_name;   //Declaration
array_name=[value1, value2,…..valueN]; //Initialization
```

The array parameter contains the list of integers and strings. It is recommended to use **an array literal** for creating an array instead of using the **new keyword**.
The new keyword only complicates the code, and sometimes it produces unexpected results. If you specify a single numeric parameter in the constructor of an array (or creating an array with **the new** keyword), then it will be treated as the length of an array.

```
var num = new Array(5); // This single numeric value indicates the size of array.
var i;
for(i=0;i<num.length;i++){
num[i]=i*5;
console.log(num[i]);
}
```

ES6 Arrays

JavaScript Arrays

JavaScript supports the following categories of arrays.

- Multidimensional array
- Passing arrays to functions
- Return array from functions

ES6 Multidimensional Arrays

ES6 also supports the multidimensional array concept. A multidimensional array can be defined as an array reference to another array for its value.

Multidimensional arrays are not directly provided in JavaScript. If you need to create a multidimensional array, you have to do it by using the one-dimensional array.

We can also say that a two-dimensional array is the simplest form of a multidimensional array.

```
var multi = [[2,3,4],[4,9,16]]

console.log(multi[0][0])

console.log(multi[0][1])

console.log(multi[0][2])

console.log(multi[1][0])

console.log(multi[1][1])

console.log(multi[1][2])
```

ES6 Array methods

Array Methods:

S.no.	Methods	Description
1.	Array.from()	It converts array-like values and iterable values into arrays.
2.	Array.of()	It creates an instance from a variable number of arguments instead of the number of arguments or type of arguments.
3.	Array.prototype.copyWithin()	It copies the part of an array to a different location within the same array.
4.	Array.prototype.find()	It finds a value from an array, based on the specific criteria that are passed to this method.
5.	Array.prototype.findIndex()	The Array.prototype.findIndex() returns the index of the first element of the given array that satisfies the given condition.
6.	Array.prototype.entries()	It returns an array iterator object, which can be used to loop through keys and values of arrays.
7.	Array.prototype.keys()	It returns an array iterator object along with the keys of the array.
8.	Array.prototype.values()	it provides the value of each key.
9.	Array.prototype.fill()	It fills the specified array elements with a static value

ES6 Array methods

Array.from() - The general function of this method is to enable new array creation from an array-like object. It converts array-like values and iterable values (such as **set** and **map**) into arrays.

Array.from(object, mapFunction, thisValue)

- **object:** This parameter value is always required. It is the object to convert to an array.
- **mapFunction:** It is optional. It is a map function to call on each item of the array.
- **thisValue:** It is also optional. It is a value to use as **this** when executing the **mapFunction**.

```
let name = Array.from(‘AmitKumar')

console.log(name)
```

Array.of(<value>) – it will create an array only with that value instead of creating the array of that size.

```
let name = Array.of(42)

console.log(name)

console.log(name.length)
```

Array.prototype.copyWithin() - This method copies the part of an array to a different location within the same array. It returns the modified array without any modification in its length.
array.copyWithin(target, start, end)

- **target:** It is always required. It is the index position to copy the elements.
- **start:** It is an optional parameter. It refers to the index position to start copying the elements. Its default value is 0. If the value of this parameter is negative, then start will be counted from the end.
- **end:** It is also an optional parameter. It refers to the index position to stop copying the elements. Its default value is the length of the array.

ES6 Array methods

Array.prototype.copyWithin() - This method copies the part of an array to a different location within the same array. It returns the modified array without any modification in its length.

array.copyWithin(target, start, end)

- **target:** It is always required. It is the index position to copy the elements.
- **start:** It is an optional parameter. It refers to the index position to start copying the elements. Its default value is 0. If the value of this parameter is negative, then start will be counted from the end.
- **end:** It is also an optional parameter. It refers to the index position to stop copying the elements. Its default value is the length of the array.

```
const num = [1,2,3,4,5,6,7,8,9,10];
const num1 = [1,2,3,4,5,6,7,8,9,10];
const num2 = [1,2,3,4,5,6,7,8,9,10];
console.log(num.copyWithin(1,3,5));
console.log(num1.copyWithin(1,3)); //omitting the parameter end
console.log(num2.copyWithin(1)); //omitting the parameters start and end
```

Array.prototype.find() - It finds a value from an array, based on the specific criteria that are passed to this method. It returns the first element value that satisfies the given condition.

array.find(callback(currentValue, index, arr),thisValue)

- **callback:** It represents the function that executes every element.
- **currentValue:** It is the required parameter. It is the value of the current element.
- **index:** it is an optional parameter. It is the array index of the current element.
- **arr:** It is also an optional parameter. It is the array on which the find() operated.
- **thisValue:** It is optional. It is a value that is used as **this** while using callback.

```
var arr=[5,22,19,25,34];
var result=arr.find(x=>x>20);
console.log(result);
```

ES6 Array methods

Array.prototype.findIndex()

The Array.prototype.findIndex() method returns the index of the first element of the given array that satisfies the given condition. If no element satisfies the condition, then it returns -1.

```
array.findIndex(callback(value,index,arr),thisArg)

var arr=[5,22,19,25,34];
var result=arr.findIndex(x=>x>20);
console.log(result)
```

Array.prototype.entries()

This method returns an array iterator object, which can be used to loop through keys and values of arrays.
Entries will return an array of arrays, in which every child array is an array of [index, value].

```
var colours = ["Red", "Yellow", "Blue", "Black"];
var show = colours.entries();

for (i of show) {
  console.log(i);
}
```

Array.prototype.keys()

This method works similarly to the **Array.entries() method**. As its name implies, it is used to return an array iterator object along with the keys of the array.

```
var colours = ["Red", "Yellow", "Blue", "Black"];
var show = colours.keys();
console.log(...show);
```

ES6 Array methods

Array.prototype.values()

This method is similar to **Array.keys()** and **Array.entries()** except that it provides the value of each key.

```
var colours = ["Red", "Yellow", "Blue", "Black"];

var show = colours.values();

console.log(...show);
```

Array.prototype.fill()

This method fills the specified array elements with a static value. The value can be used to fill a part of an array or to fill the entire array. It modifies the original array. You can specify the position of the start and end the filling. If not specified, then all elements will be filled.

```
array.fill(value, start, end)
```

Parameter Values

value: It is a static value to fill the array. It is always required.

start: It is the index to start filling the array. It is optional, and its default value is 0.

end: It is the index to stop filling the array. It is also optional, and its default value is the length of the array.

```
var colours = ["Red", "Yellow", "Blue", "Black"];

var show = colours.fill("Green",2,4);

console.log(...show);
```

ES6 Array de-structuring

Destructuring means to break down a complex structure into simpler parts. With the syntax of destructuring, you can extract smaller fragments from objects and arrays. It can be used for assignments and declaration of a variable.

Destructuring is an efficient way to extract multiple values from data that is stored in arrays or objects. When destructuring an array, we use their positions (or index) in an assignment.

```
var arr = ["Hello", "World"]
// destructuring assignment
var [first, second] = arr;
console.log(first); // Hello
console.log(second); // World

var colors = ["Violet", "Indigo", "Blue", "Green", "Yellow", "Orange", "Red"];
// destructuring assignment
var[color1, color2, color3] = colors;
console.log(color1); // Violet
console.log(color2); // Indigo
console.log(color3); // Blue

var colors = ["Violet", "Indigo", "Blue", "Green", "Yellow", "Orange", "Red"];
// destructuring assignment
var[color1, ,color3, ,color5] = colors; //Leave space for unpick elements
console.log(color1); // Violet
console.log(color3); // Blue
console.log(color5); // Yellow
```

ES6 Array de-structuring

Array destructuring and Rest operator

By using the rest operator (…) in array destructuring, you can put all the remaining elements of an array in a new array.

```
var colors = ["Violet", "Indigo", "Blue", "Green", "Yellow", "Orange", "Red"];

// destructuring assignment
var [a,b,...args] = colors;
console.log(a);
console.log(b);
console.log(args);
```

Array destructuring and Default values

if you are taking a value from the array and that value is undefined, then you can assign a default value to a variable.

```
var x, y;

[x=50, y=70] = [100];
console.log(x); // 100
console.log(y); // 70
```

Swapping Variables

The values of the two variables can be swapped in one destructuring expression. The array destructuring makes it easy to swap the values of variables without using any temporary variable.

```
var x = 100, y = 200;
[x, y] = [y, x];
console.log(x); // 200
console.log(y); // 100
```

ES6 Array de-structuring

Parsing returned array from functions

A function can return an array of values. It is always possible to return an array from a function, but array destructuring makes it more concise to parse the returned array from functions.

```
function array() {
   return [100, 200, 300];
}

var [x, y, z] = array();

console.log(x); // 100
console.log(y); // 200
console.log(z); // 300
```

Array destructuring and Default values

if you are taking a value from the array and that value is undefined, then you can assign a default value to a variable.

```
var x, y;

[x=50, y=70] = [100];
console.log(x); // 100
console.log(y); // 70
```

Swapping Variables

The values of the two variables can be swapped in one destructuring expression. The array destructuring makes it easy to swap the values of variables without using any temporary variable.

```
var x = 100, y = 200;
[x, y] = [y, x];
console.log(x); // 200
console.log(y); // 100
```

Array.forEach Method

```
array.forEach(function(currentValue, index, arr))
```

The forEach method executes a provided function once for every element in the array.

- function(currentValue, index, arr) - a function to be run for each element of an array
- currentValue - the value of an array
- index (optional) - the index of the current element
- arr (optional) - the array of the current elements

```
let students = ['John', 'Sara', 'Jack'];

// using forEach
students.forEach(myFunction);

function myFunction(item, index, arr) {

    // adding strings to the array elements
    arr[index] = 'Hello ' + item;
}
console.log(students);
```

Here, inside the forEach loop callback function, each element of the array is automatically passed as the first parameter of the function.

Array.forEach Method

The equivalent for loop code for the above example looks like this:

```
const months = ['January', 'February', 'March', 'April'];

for(let i = 0; i < months.length; i++) {
  console.log(months[i]);
}

/* output
January
February
March
April
*/
```

The thing you need to keep in mind is that the forEach method does not return any value.
Take a look at the below code:

```
const months = ['January', 'February', 'March', 'April'];
const returnedValue = months.forEach(function (month) {
  return month;
});

console.log('returnedValue: ', returnedValue); // undefined
```

Array.forEach Method

Note that forEach is only used to loop through the array and perform some processing or logging. It does not return any value, even if you explicitly return a value from the callback function (this means that the returned value comes as undefined in the above example).
In all the above examples, we have used only the first parameter of the callback function. But the callback function also receives two additional parameters, which are:

index - the index of the element which is currently being iterated
array - original array which we're looping over

```
const months = ['January', 'February', 'March', 'April'];

months.forEach(function(month, index, array) {
  console.log(month, index, array);
});

/* output

January 0 ["January", "February", "March", "April"]
February 1 ["January", "February", "March", "April"]
March 2 ["January", "February", "March", "April"]
April 3 ["January", "February", "March", "April"]

*/
```

Depending on the requirement, you may find it useful to use the index and array parameters.

Array.forEach Method

ForEach with Map:

```
let map = new Map();
// inserting elements
map.set('name', 'Jack');
map.set('age', '27');

// looping through Map
map.forEach (myFunction);

function myFunction(value, key) {

    console.log(key + '- ' + value);
}
```

Advantages of using forEach instead of a for loop:

- Using a forEach loop makes your code shorter and easier to understand
- When using a forEach loop, we don't need to keep track of how many elements are available in the array. So it avoids the creation of an extra counter variable.
- Using a forEach loop makes code easy to debug because there are no extra variables for looping through the array
- The forEach loop automatically stops when all the elements of the array are finished iterating.

Browser Support
All modern browsers and Internet Explorer (IE) version 9 and above
Microsoft Edge version 12 and above

Array.map Method

The Array map method is the most useful and widely used array method among all other methods. The map method executes a provided function once for every element in the array and it returns a new transformed array.
The Array.map method has the following syntax:

```
arr.map(callback(currentValue), thisArg)
```

map() Parameters

- callback - The function called for every array element. Its return values are added to the new array.
  It takes in:
  - currentValue - The current element being passed from the array.
- thisArg (optional) - Value to use as this when executing callback. By default, it is undefined.

map() Return Value

- Returns a new array with elements as the return values from the callback function for each element.

Notes:

- map() does not change the original array.
- map() executes callback once for each array element in order.
- map() does not execute callback for array elements without values.

Take a look at the below code:

```
const months = ['January', 'February', 'March', 'April'];
const transformedArray = months.map(function (month) {
  return month.toUpperCase();
});
console.log(transformedArray); // ["JANUARY", "FEBRUARY", "MARCH", "APRIL"]
```

In the above code, inside the callback function, we’re converting each element to uppercase and returning it.

Array.map Method

The equivalent for loop code for the above example looks like this:

```
const months = ['January', 'February', 'March', 'April'];
const converted = [];

for(let i = 0; i < months.length; i++) {
 converted.push(months[i].toUpperCase());
};

console.log(converted); // ["JANUARY", "FEBRUARY", "MARCH", "APRIL"]
```

Using map helps to avoid creating a separate converted array beforehand for storing the converted elements. So it saves memory space and also the code looks much cleaner using array map, like this:

```
const months = ['January', 'February', 'March', 'April'];

console.log(months.map(function (month) {
  return month.toUpperCase();
})); // ["JANUARY", "FEBRUARY", "MARCH", "APRIL"]
```

Note that the map method returns a new array that is of the exact same length as the original array.

Array.map Method

The difference between the forEach and map methods is that forEach is only used for looping and does not return anything back. On the other hand, the map method returns a new array that is of the exact same length as the original array.

Also, note that map does not change the original array but returns a new array.
Take a look at the below code:

```
const users = [
  {
    first_name: 'Mike',
    last_name: 'Sheridan'
  },
  {
    first_name: 'Tim',
    last_name: 'Lee'
  },
  {
    first_name: 'John',
    last_name: 'Carte'
  }
];
const usersList = users.map(function (user) {
  return user.first_name + ' ' + user.last_name;
});

console.log(usersList); // ["Mike Sheridan", "Tim Lee", "John Carte"]
```

Array.map Method

Here, by using the array of objects and map methods, we're easily generating a single array with first and last name concatenated.

In the above code, we're using the + operator to concatenate two values. But it's much more common to use ES6 template literal syntax as shown below:

```
const users = [
  {
    first_name: 'Mike',
    last_name: 'Sheridan'
  },
  {
    first_name: 'Tim',
    last_name: 'Lee'
  },
  {
    first_name: 'John',
    last_name: 'Carte'
  }
];

const usersList = users.map(function (user) {
  return `${user.first_name} ${user.last_name}`;
});

console.log(usersList); // ["Mike Sheridan", "Tim Lee", "John Carte"]
```

Array.map Method

The array map method is also useful, if you want to extract only specific data from the array like this:

```
const users = [
  {
    first_name: 'Mike',
    last_name: 'Sheridan',
    age: 30
  },
  {
    first_name: 'Tim',
    last_name: 'Lee',
    age: 45
  },
  {
    first_name: 'John',
    last_name: 'Carte',
    age: 25
  }
];
const surnames = users.map(function (user) {
  return user.last_name;
});
console.log(surnames); // ["Sheridan", "Lee", "Carte"]
```

In the above code, we're extracting only the last names of each user and storing them in an array.

Array.map Method

We can even use map to generate an array with dynamic content as shown below:

```
const users = [
  {
    first_name: 'Mike',
    location: 'London'
  },
  {
    first_name: 'Tim',
    location: 'US'
  },
  {
    first_name: 'John',
    location: 'Australia'
  }
];

const usersList = users.map(function (user) {
  return `${user.first_name} lives in ${user.location}`;
});

console.log(usersList); // ["Mike lives in London", "Tim lives in US", "John lives in Australia"]
```

Note that in the above code, we're not changing the original users array. We're creating a new array with dynamic content because map always returns a new array.

Array.map Method

Advantages of using the map method

- It helps quickly generate a new array without changing the original array
- It helps generate an array with dynamic content based on each element
- It allows us to quickly extract any element of the array
- It generates an array with the exact same length as the original array

Browser Support:

All modern browsers and Internet Explorer (IE) version 9 and above
Microsoft Edge version 12 and above

Array.filter Method

The filter method returns a new array with all the elements that satisfy the provided test condition.

```
arr.filter(callback(element), thisArg)
```

filter() Parameters

- callback - The test function to execute on each array element; returns true if element passes the test, else false. It takes in:
    - element - The current element being passed from the array.
- thisArg (optional) - The value to use as this when executing callback. By default, it is undefined.

filter() Return Value

- Returns a new array with only the elements that passed the test.

The filter method takes a callback function as the first argument and executes the callback function for every element of the array. Each array element value is passed as the first parameter to the callback function.

```
const employees = [
  { name: 'David Carlson', age: 30 }, { name: 'John Cena', age: 34 }, { name: 'Mike Sheridan', age: 25 }, { name: 'John Carte', age: 50 }
];
const employee = employees.filter(function (employee) {
  return employee.name.indexOf('John') > -1;
});
console.log(employee); // [ { name: "John Cena", age: 34 }, { name: "John Carte", age: 50 }]
```

As can be seen in the above code, using filter helps to find all the elements from the array that match the specified test condition.

Array.filter Method

So using filter does not stop when it finds a particular match but keeps checking for other elements in the array that match the condition. Then it returns all the matching elements from the array.

The main difference between find and filter is that find only returns the first matching element of the array, but using filter returns all the matching elements from the array.
Note that the filter method always returns an array. If no element passes the test condition, an empty array will be returned.

The equivalent for loop code for the above example looks like this:

```
const employees = [
  { name: 'David Carlson', age: 30 },
  { name: 'John Cena', age: 34 },
  { name: 'Mike Sheridan', age: 25 },
  { name: 'John Carte', age: 50 }
];

let filtered = [];
for(let i = 0; i < employees.length; i++) {
 if(employees[i].name.indexOf('John') > -1) {
   filtered.push(employees[i]);
 }
}
console.log(filtered); // [ { name: "John Cena", age: 34 }, { name: "John Carte", age: 50 }]
```

Array.filter Method

Advantages of using the filter method

- It allows us to quickly find all the matching elements from the array
- It always returns an array even if there is no match, so it avoids writing extra if conditions
- It avoids the need of creating an extra variable to store the filtered elements

Browser Support:

All modern browsers and Internet Explorer (IE) version 9 and above
Microsoft Edge version 12 and above

The Array.reduce Method

arr.reduce(callback(accumulator, currentValue), initialValue)

reduce() Parameters

- callback - The function to execute on each array element (except the first element if no initialValue is provided).

It takes in

- accumulator - It accumulates the callback's return values.
- currentValue - The current element being passed from the array.
- initialValue (optional) - A value that will be passed to callback() on first call. If not provided, the first element acts as the accumulator on the first call and callback() won't execute on it.

Note: Calling reduce() on an empty array without initialValue will throw TypeError.

reduce() Return Value

- Returns the single value resulting after reducing the array.

Notes:

- reduce() executes the given function for each value from left to right.
- reduce() does not change the original array.
- It is almost always safer to provide initialValue.

The reduce method executes a reducer function (that you provide) on each element of the array, resulting in a single output value.

Note that the output of the reduce method is always a single value. It can be an object, a number, a string, an array, and so on. It depends on what you want the output of reduce method to generate but it's always a single value.

The Array.reduce Method

Suppose that you want to find the sum of all the numbers in the array. You can use the reduce method for that.

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce(function(accumulator, number) {
  return accumulator + number;
}, 0);
console.log(sum); // 15
```

The reduce method accepts a callback function that receives accumulator, number, index and array as the values. In the above code, we're using only accumulator and number.
The accumulator will contain the initialValue to be used for the array. The initialValue decides the return type of the data returned by the reduce method.
The number is the second parameter to the callback function that will contain the array element during each iteration of the loop.
In the above code, we have provided 0 as the initialValue for the accumulator. So the first time the callback function executes, the accumulator + number will be 0 + 1 = 1 and we're returning back the value 1.

The next time the callback function runs, accumulator + number will be 1 + 2 = 3 (1 here is the previous value returned in the last iteration and 2 is the next element from the array).

Then, the next time the callback function runs, accumulator + number will be
3 + 3 = 6(the first 3 here is the previous value returned in the last iteration and the next 3 is the next element from the array) and it will continue this way until all the elements in the numbers array are not iterated.

So the accumulator will retain the value of the last operation just like a static variable.
In the above code, initialValue of 0 is not required because all the elements of the array are integers.

The Array.reduce Method

So the below code will also work:

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce(function (accumulator, number) {
  return accumulator + number;
});

console.log(sum); // 15
```

Here, the accumulator will contain the first element of the array and number will contain the next element of the array (1 + 2 = 3 during the first iteration and then 3 + 3 = 6 during the next iteration, and so on).

But it’s always good to specify the initial Value of accumulator as it makes it easy to understand the return type of the reduce method and get the correct type of data back.

Take a look at the below code:

```
const numbers = [1, 2, 3, 4, 5];
const doublesSum = numbers.reduce(function (accumulator, number) {
  return accumulator + number * 2;
}, 10);

console.log(doublesSum); // 40
```

The Array.reduce Method

Here, we’re multiplying each element of the array by 2. We have provided an initialValue of 10 to the accumulator so 10 will be added to the final result of the sum like this:

```
[1 * 2, 2 * 2, 3 * 2, 4 * 2, 5 * 2] = [2, 4, 6, 8, 10] = 30 + 10 = 40
```

Suppose, you have an array of objects with x and y coordinates and you want to get the sum of x coordinates. You can use the reduce method for that.

```
const coordinates = [
  { x: 1, y: 2 },
  { x: 2, y: 3 },
  { x: 3, y: 4 }
];
const sum = coordinates.reduce(function (accumulator, currentValue) {
    return accumulator + currentValue.x;
}, 0);
Console.log(sum); // 6
```

Remove Duplicates:

```
let ageGroup = [18, 21, 1, 1, 51, 18, 21, 5, 18, 7, 10];
let uniqueAgeGroup = ageGroup.reduce(function (accumulator, currentValue) {
  if (accumulator.indexOf(currentValue) === -1) {
    accumulator.push(currentValue);
  }
  return accumulator;
}, []);
console.log(uniqueAgeGroup); // [ 18, 21, 1, 51, 5, 7, 10 ]
```

The Array.reduce Method

Advantages of using the reduce method

- Using reduce allows us to generate any type of simple or complex data based on the array
- It remembers the previously returns data from the loop so helps us avoid creating a global variable to store the previous value

Browser Support:

All modern browsers and Internet Explorer (IE) version 9 and above

Microsoft Edge version 12 and above

The Array.slice Method

The slice() method returns a shallow copy of a portion of an array into a new array object.

Example:

```
let numbers = [2, 3, 5, 7, 11, 13, 17];

// create another array by slicing numbers from index 3 to 5
let newArray = numbers.slice(3, 6);
console.log(newArray);

// Output: [ 7, 11, 13 ]
```

slice() Syntax
The syntax of the slice() method is:

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

Here, arr is an array.

slice() Parameters
The slice() method takes in:

start (optional) - Starting index of the selection. If not provided, the selection starts at start 0.
end (optional) - Ending index of the selection (exclusive). If not provided, the selection ends at the index of the last element.
slice() Return Value
Returns a new array containing the extracted elements.

The Array.slice Method

Example 1: JavaScript slice() method

```
let languages = ["JavaScript", "Python", "C", "C++", "Java"];

// slicing the array (from start to end)
let new_arr = languages.slice();
console.log(new_arr); // [ 'JavaScript', 'Python', 'C', 'C++', 'Java' ]

// slicing from the third element
let new_arr1 = languages.slice(2);
console.log(new_arr1); // [ 'C', 'C++', 'Java' ]

// slicing from the second element to fourth element
let new_arr2 = languages.slice(1, 4);
console.log(new_arr2); // [ 'Python', 'C', 'C++' ]
```

Output:

```
[ 'JavaScript', 'Python', 'C', 'C++', 'Java' ]
[ 'C', 'C++', 'Java' ]
[ 'Python', 'C', 'C++' ]
```

The Array.slice Method

Example 2: JavaScript slice() With Negative index

In JavaScript, you can also use negative start and end indices. The index of the last element is -1, the index of the second last element is -2, and so on.

```
const languages = ["JavaScript", "Python", "C", "C++", "Java"];

// slicing the array from start to second-to-last
let new_arr = languages.slice(0, -1);
console.log(new_arr); // [ 'JavaScript', 'Python', 'C', 'C++' ]

// slicing the array from third-to-last
let new_arr1 = languages.slice(-3);
console.log(new_arr1); // [ 'C', 'C++', 'Java' ]
```

Output

```
[ 'JavaScript', 'Python', 'C', 'C++' ]
[ 'C', 'C++', 'Java' ]
```

ES6 String

JavaScript string is an object which represents the sequence of characters. Generally, strings are used to hold text-based values such as a person name or a product description.

In JavaScript, any text within the single or double quotes is considered as a string. There are two ways for creating a string in JavaScript:

- By using a string literal
- By using string object (using the new keyword)

By using a string literal
The string literals can be created by using either double quotes or by using single quotes.

```
var stringname = "string value";
```

By using String object (using the new keyword)

```
var stringname = new String ("string literal");
```

String Properties :

S.no.	Property	Description
1.	constructor	It returns the constructor function for an object.
2.	length	It returns the length of the string.
3.	prototype	It allows us to add the methods and properties to an existing object.

ES6 String

JavaScript string constructor property

The constructor property returns the constructor function for an object. Instead of the name of the function, it returns the reference of the function.

```
var str = new String("Hello World");
console.log("Value of str.constructor is: "+str.constructor);
```

JavaScript string length property

As its name implies, this property returns the number of characters or the length of the string.

```
var str = new String("Hello World");
console.log("The number of characters in the string str is: "+str.length);
```

JavaScript string prototype property

It allows us to add new methods and properties in an existing object type. It is a global property which is available with almost all the objects of JavaScript.

```
function student(name, qualification){
this.name = name;
this.qualification = qualification;
}
student.prototype.age = 20;
var stu = new student('Daniel Grint' , 'BCA');
console.log(stu.name);
console.log(stu.qualification);
console.log(stu.age);
```

ES6 String Method

startsWith() method
It is **a case-sensitive** method, which determines whether the string begins with the specified string characters or not. It returns true if the string begins with the characters and returns false if not.

string.startsWith(searchValue, startPosition) :

- **searchValue:** It is the required parameter of this method.
  It includes the characters to be searched for at the start of the string.
- **startPosition:** It is an optional parameter. Its default value is **0**.
  It specifies the position in the string at which to begin searching.

```
var str = 'Welcome to start with method :)';
console.log(str.startsWith('Wel',0));
console.log(str.startsWith('wel',0));
```

S.no.	Methods	Description
1.	startsWith	It determines whether a string begins with the characters of a specified string.
2.	endsWith	It determines whether a string ends with the characters of a specified string.
3.	includes	It returns true if the specified argument is in the string.
4.	repeat	It returns a new string repeated based on specified count arguments.

endsWith() method :
It is also a **case-sensitive** method that determines whether a string ends with the characters of the specified string or not.

string.endsWith(searchvalue, length)

- **searchValue:** It is the required parameter that represents the characters to be searched at the end of the string.
- **length:** It is an optional parameter. It is the length of the string that will be searched. If this parameter is omitted, then the method will search in the full length of the string.

```
var str = "Welcome to ends with method.";
console.log(str.endsWith("to", 10))
console.log(str.endsWith("To", 10))
```

ES6 String Method

includes() method
It is a case-sensitive method that determines whether the string contains the characters of the specified string or not. It returns true if the string contains characters and returns false if not.

```
string.includes(searchValue, start)
```

- **searchValue:** It is a required parameter. It is the substring to search for.
- **start:** It represents the position where to start the searching in the string. Its default value is 0.

```
let str = "hello world"

console.log(str.includes('world',5));
console.log(str.includes('World', 11))
```

repeat() method :
It is used to build a new string that contains a specified number of copies of the string on which this method has been called.

```
string.repeat(count)
```

- **count:** It is a required parameter that shows the number of times to repeat the given string. The range of this parameter is from 0 to infinity.

```
var str = “hello world.";
console.log(str.repeat(5))
```

JavaScript Function

JavaScript functions are defined with the function keyword. You can use a function declaration or a function expression.

Function Declarations
Earlier in this tutorial, you learned that functions are declared with the following syntax:

```
function functionName(parameters) {
  // code to be executed
}
```

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are invoked (called upon).

Example:

```
function myFunction(a, b) {
  return a * b;
}
```

Function Expressions
A JavaScript function can also be defined using an expression.
A function expression can be stored in a variable:

```
const x = function (a, b) {return a * b};
let z = x(4, 3);
```

The function above is actually an anonymous function (a function without a name).
Functions stored in variables do not need function names. They are always invoked (called) using the variable name.
The function above ends with a semicolon because it is a part of an executable statement.

JavaScript Function

The Function() Constructor

As you have seen in the previous examples, JavaScript functions are defined with the function keyword.

Functions can also be defined with a built-in JavaScript function constructor called Function().

Example
```
const myFunction = new Function("a", "b", "return a * b");

let x = myFunction(4, 3);
```

You actually don't have to use the function constructor. The example above is the same as writing:

Example
```
const myFunction = function (a, b) {return a * b};

let x = myFunction(4, 3);
```

Most of the time, you can avoid using the new keyword in JavaScript.

JavaScript Function

Function Hoisting

Earlier in this tutorial, you learned about "hoisting" (JavaScript Hoisting).

Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope.

Hoisting applies to variable declarations and to function declarations.

Because of this, JavaScript functions can be called before they are declared:

```
myFunction(5);

function myFunction(y) {
  return y * y;
}
```

Functions defined using an expression are not hoisted.

JavaScript Function

Self-Invoking Functions

Function expressions can be made "self-invoking".

A self-invoking expression is invoked (started) automatically, without being called.

Function expressions will execute automatically if the expression is followed by ().

You cannot self-invoke a function declaration.

You have to add parentheses around the function to indicate that it is a function expression:

Example
```
(function () {
  let x = "Hello!!";  // I will invoke myself
})();
```

The function above is actually an anonymous self-invoking function (function without name).

JavaScript Function

Functions Can Be Used as Values
JavaScript functions can be used as values:

Example
```
function myFunction(a, b) {
  return a * b;
}

let x = myFunction(4, 3);
```

JavaScript functions can be used in expressions:

Example
```
function myFunction(a, b) {
  return a * b;
}

let x = myFunction(4, 3) * 2;
```

Functions are Objects
The typeof operator in JavaScript returns "function" for functions.

But, JavaScript functions can best be described as objects.

JavaScript Function

JavaScript functions have both properties and methods.

The arguments.length property returns the number of arguments received when the function was invoked:

Example
```
function myFunction(a, b) {
  return arguments.length;
}
```
The toString() method returns the function as a string:

Example
```
function myFunction(a, b) {
  return a * b;
}

let text = myFunction.toString();
```
A function defined as the property of an object, is called a method to the object.
A function designed to create new objects, is called an object constructor.

JavaScript Function

Arrow Functions
Arrow functions allows a short syntax for writing function expressions.

You don't need the function keyword, the return keyword, and the curly brackets.

Example
```
// ES5
var x = function(x, y) {
  return x * y;
}

// ES6
const x = (x, y) => x * y;
```

Arrow functions do not have their own this. They are not well suited for defining object methods.

Arrow functions are not hoisted. They must be defined before they are used.

Using const is safer than using var, because a function expression is always 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 them:

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

ES6 Template Literals

Template literals are a new feature introduced in ECMAScript 2015/ ES6. It provides an easy way to create multiline strings and perform string interpolation. Template literals are the string literals and allow embedded expressions.

Before ES6, template literals were called as **template strings**. Unlike quotes in strings, template literals are enclosed by the **backtick (` `)** character (key below the **ESC** key in QWERTY keyboard). Template literals can contain placeholders, which are indicated by the dollar sign and curly braces **($(expression})**. Inside the backticks, if we want to use an expression, then we can place that expression in the **($(expression})**.

```
var str = `string value`;
```

Multiline strings

In normal strings, we have to use an escape sequence **\n** to give a new line for creating a multiline string. However, in template literals, there is no need to use **\n** because string ends only when it gets **backtick(`)** character.

```
// Without template literal
console.log('Without template literal \n multiline string');

// With template literal
console.log(`Using template literal
multiline string`);
```

String Interpolation

ES6 template literals support string interpolation. Template literals can use the placeholders for string substitution. To embed expressions with normal strings, we have to use the **${}** syntax.

```
var name = 'World';
var cname = ’Template Literals';
console.log(`Hello, ${name}!
Welcome to ${cname}`);
```

OOPS

Object Orientation is a software development paradigm that follows real-world modelling. Object Orientation, considers a program as a collection of objects that communicates with each other via mechanism called methods. ES6 supports these object-oriented components too.

Object-Oriented Programming Concepts:

- Object - An object is a real-time representation of any entity. According to Grady Brooch, every object is said to have 3 features
    - State – attributes of an object
    - Behavior – how the object will act.
    - Identity – A unique value that distinguishes an object
- Class – blueprint for creating objects. A class encapsulate data for the object.
- Method – Methods facilitate communication between objects
- Declaring a class - Class A { }
- Class expression:

```
var Polygon = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

OOPS

Creating Objects:

var object_name= new class_name([arguments])

- The new keyword is responsible for instantiation.
- The right hand side of the expression invokes the constructor. The constructor should be passed values if it is parameterized.
- E.g. - var obj = new Polygon(10,12)

Accessing Functions:

```
class Polygon {
   constructor(height, width) { this.h = height; this.w = width; }
   test() {
      console.log("The height of the polygon: ", this.h) ;
      console.log("The width of the polygon: ",this. w)
   } }
//creating an instance
var polyObj = new Polygon(10,20);
polyObj.test();
```

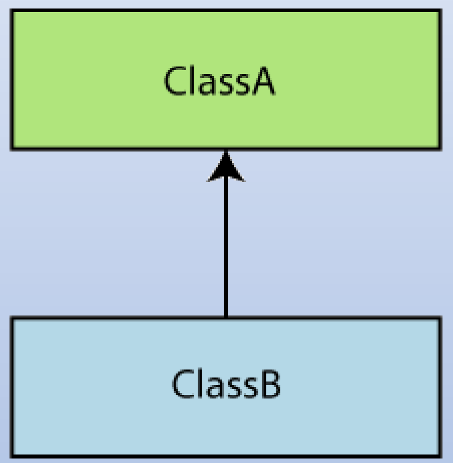
OOPS

Setter and Getters:

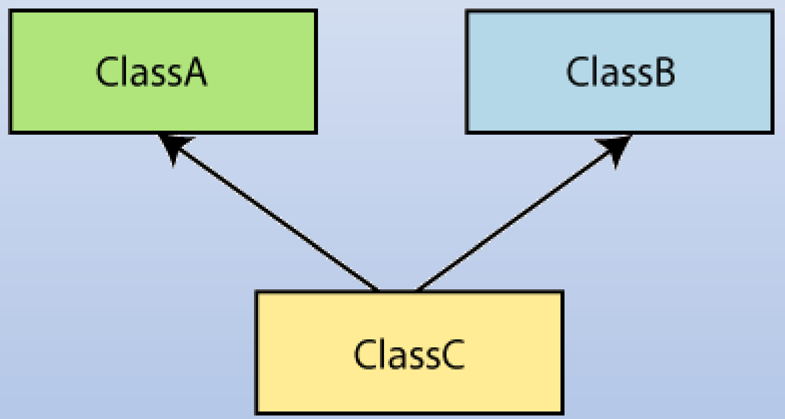
```
<script>
  class Student {
    constructor(rno,fname,lname){
      this.rno = rno
      this.fname = fname
      this.lname = lname
      console.log('inside constructor')
    }
    get fullName(){
      console.log('inside getter')
      return this.fname + " - "+this.lname
    }
  }
  let s1 = new Student(101,'Sachin','Tendulkar')
  console.log(s1)
  //getter is called
  console.log(s1.fullName)
</script>
```

OOPS

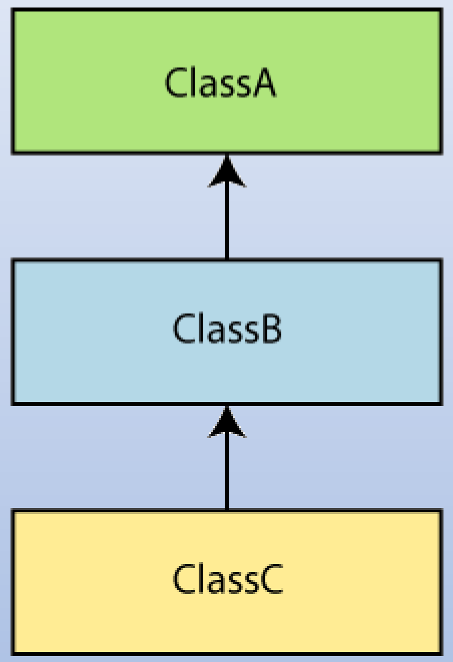
Inheritance:



Single Inheritance



Multiple Inheritance



Multilevel Inheritance

OOPS

Method Overriding:

It is a feature that allows a child class to provide a specific implementation of a method which has been already provided by one of its parent class.

There are some rules defined for method overriding -

The method name must be the same as in the parent class.

Method signatures must be the same as in the parent class.

```
class Parent {
  show() {
    console.log("It is the show() method from the parent class");
  } }
class Child extends Parent {
  show() {
    console.log("It is the show() method from the child class");
  }  }
var obj = new Child();
obj.show();
```

OOPS

The Super Keyword:

It allows the child class to invoke the properties, methods, and constructors of the immediate parent class.

Method signatures must be the same as in the parent class.

```
class Parent {
  show() {
    console.log("It is the show() method from the parent class");
  } }
class Child extends Parent {
  show() {
    super.show();
    console.log("It is the show() method from the child class");
  }  }
var obj = new Child();
obj.show();
```

Objects

Objects are the collection of key/value pairs that can be modified throughout the lifecycle of an object as similar to hash map or dictionary. Objects allow us to define custom data types in JavaScript.

Unlike primitive data types, we can represent complex or multiple values using objects. The values can be the array of objects or the scalar values or can be the functions. Data inside objects are unordered, and the values can be of any type.

An object can be created by using curly braces **{...}** along with an optional properties list. The property is a **"key:value"** pair, where the key is a string or a property name, and the value can be anything.

There are two syntaxes to create an empty object:

By using object literal

By using object constructor

```
1. var user =  {};  // 'object literal' syntax
2. var name = new Object();  //'object constructor' syntax
```

Objects

Merge objects in ES6

It is possible to merge two JavaScript objects in ES6 by using two methods, which are listed below:

- Object.assign() method
- Object spread syntax method

By using Object.assign() method

This method is used for copying the values and properties from one or more source object to a target object. It returns the target object, including properties and values copied from the target object.

```
Object.assign(target, sources)
var obj1 = {1 : "Hello", 2: "World"};
 var obj2 = { 3 : "Welcome", 4: "to"};
 var obj3 = { 5 : “Objects"}
 // Using Object.assign()
var final_obj = Object.assign(obj1, obj2, obj3);
console.log(final_obj);
```

Objects

Object Cloning

Cloning is the process of copying an object from one variable to another variable. We can clone an object by using the **object.assign() method**:

```
 let obj1 = {
 name: 'Anil',
 age: 22
};
let cloneobj = Object.assign({}, obj1);
cloneobj.age = 32;
console.log(obj1);
console.log(cloneobj);
```

Objects

By using Object spread syntax

It is widely used in a variable array where multiple values are expected. As the objects in JavaScript are key-value paired entities, we can merge them into one by using the spread operator.

```
 var new_obj = [...obj1, ...obj2, ...]

 var obj1 = {1 : "Hello", 2: "World"};

var obj2 = { 3 : "Welcome", 4: "to"};

var obj3 = { 5 : “spread Syntax"}


var final_obj = {...obj1, ...obj2, ...obj3};

console.log(final_obj);
```

Objects

Delete properties

It can be possible to remove or delete a property by using the delete operator. Let us understand how to remove a property by using the following example.

```
var obj = new Object;

obj.a = 50;

obj.b = 200;


delete obj.a;

console.log (obj.a);
```

Object Destructuring

It is similar to array destructuring except that instead of values being pulled out of an array, the properties (or keys) and their corresponding values can be pulled out from an object.

When destructuring the objects, we use keys as the name of the variable. The variable name must match the property (or keys) name of the object. If it does not match, then it receives an **undefined** value. This is how JavaScript knows which property of the object we want to assign.

In object destructuring, the values are extracted by the keys instead of position (or index).

```
const num = {x: 100, y: 200};

const {x, y} = num;

console.log(x); // 100

console.log(y); // 200


const student = {name: 'Arun', position: 'First', rollno: '24'};

const {name, position, rollno} = student;

console.log(name); // Arun

console.log(position); // First

console.log(rollno); // 24
```

Object Destructuring

Object destructuring and default values:

Like array destructuring, a default value can be assigned to the variable if the value unpacked from the object is **undefined**.

```
const {x = 100, y = 200} = {x: 500};
console.log(x); // 500
console.log(y); // 200
```

Assigning new variable names:

We can assign a variable with a different name than the property of the object..

```
const num = {x: 100, y: 200};
const {x: new1, y: new2} = num;
console.log(new1); //100
console.log(new2); //200
```

Object Destructuring

Assignment without declaration:

if the value of the variable is not assigned when you declare it, then you can assign its value during destructuring..

```
let name, division;
({name, division} = {name: 'Anil', division: 'First'});
console.log(name); // Anil
console.log(division); // First
```

Object destructuring and rest operator:

By using the rest operator (…) in object destructuring, we can put all the remaining keys of an object in a new object variable.

```
let {a, b, ...args} = {a: 100, b: 200, c: 300, d: 400, e: 500}
console.log(a);
console.log(b);
console.log(args);
```

ES6 Map

ES6 is a series of new features that are added to the JavaScript. Prior to ES6, when we require the mapping of keys and values, we often use an object. It is because the object allows us to map a key to the value of any type.

ES6 provides us a new collection type called **Map**, which holds the key-value pairs in which values of any type can be used as either keys or values. A Map object always remembers the actual insertion order of the keys. Keys and values in a Map object may be primitive or objects. It returns the new or empty Map.

Maps are ordered, so they traverse the elements in their insertion order.

```
var map = new Map([iterable]);
```

The **Map ()** accepts an optional iterable object, whose elements are in the key-value pairs.

- Map.prototype.size
- It returns the number of elements in the Map object.

```
var map = new Map();
   map.set('John', 'author');
   map.set('arry', 'publisher');
   map.set('Mary', 'subscriber');
   map.set('James', 'Distributor');
console.log(map.size);
```

ES6 Map

Map Methods

The Map object includes several methods, which are tabulated as follows:

S.no.	Methods	Description
1.	Map.prototype.clear()	It removes all the keys and values pairs from the Map object.
2.	Map.prototype.delete(key)	It is used to delete an entry.
3.	Map.prototype.has(value)	It checks whether or not the corresponding key is in the Map object.
4.	Map.prototype.entries()	It is used to return a new iterator object that has an array of key-value pairs for every element in the Map object in insertion order.
5.	Map.prototype.forEach(callbackFn[, thisArg])	It executes the **callback** function once, which is executed for each element in the Map.
6.	Map.prototype.keys()	It returns an iterator for all keys in the Map.
7.	Map.prototype.values()	It returns an iterator for every value in the Map.

ES6 Map

Weak Maps

Weak Maps are almost similar to normal Maps except that the keys in weak maps must be objects. It stores each element as a key-value pair where keys are weakly referenced. Here, the keys are objects, and the values are arbitrary values. A Weak Map object only allows the keys of an object type. If there is no reference to a key object, then they are targeted to garbage collection. In weak Map, the keys are not enumerable. So, there is no method to get the list of keys.

A weak map object iterates its elements in the insertion order. It only includes **delete(key), get(key), has(key)** and **set(key, value)** method.

```
'use strict'

let wp = new WeakMap();

let obj = {};

console.log(wp.set(obj,"Welcome to Map"));

console.log(wp.has(obj));
```

Output:

```
WeakMap { <items unknown> }

true
```

ES6 Map

The for...of loop and Weak Maps

The for...of loop is used to perform an iteration over keys, values of the Map object. The following example will illustrate the traversing of the Map object by using a for...of loop.

```
'use strict'
var colors = new Map([
  ['1', 'Red'],
  ['2', 'Green'],
  ['3', 'Yellow'],
  ['4', 'Violet']
]);
for (let col of colors.values()) {
   console.log(col);
}
console.log(" ")
for(let col of colors.entries())
console.log(`${col[0]}: ${col[1]}`);
```

ES6 Map

Iterator and Map

An iterator is an object, which defines the sequence and a return value upon its termination. It allows accessing a collection of objects one at a time. Set and Map both include the methods that return an iterator.

Iterators are the objects with the **next()** method. When the **next()** method gets invoked, the iterator returns an object along with the **"value"** and **"done"** properties.

```
'use strict'
var colors = new Map([
  ['1', 'Red'],
  ['2', 'Green'],
  ['3', 'Yellow'],
  ['4', 'Violet']
]);
var itr = colors.values();  //var itr = colors.entries();  // var itr = colors.keys();
console.log(itr.next());
console.log(itr.next());
console.log(itr.next());
```

ES6 Set

A set is a data structure that allows you to create a collection of unique values. Sets are the collections that deal with single objects or single values.

Set is the collection of values similar to arrays, but it does not contain any duplicates. It allows us to store unique values. It supports both primitive values and object references.

As similar to maps, sets are also ordered, i.e., the elements in sets are iterated in their insertion order. It returns the set object.

```
var s = new Set("val1","val2","val3");
```

example:

```
let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
console.log(colors);
console.log(colors.size);
let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
colors.add('Violet');
colors.add('Indigo');
colors.add('Blue');
colors.add('Violet');
console.log(colors.size);
console.log(colors);
```

ES6 Set

Set Methods

S.no.	Methods	Description
1.	**Set.prototype.add(value)**	It appends a new element to the given value of the set object.
2.	**Set.prototype.clear()**	It removes all the elements from the set object.
3.	**Set.prototype.delete(value)**	It removes the element which is associated with the corresponding value.
4.	**Set.prototype.entries()**	It returns a new iterator object, which contains an array of each element in the Set object in insertion order.
5.	**Set.prototype.forEach(callbackFn[, thisArg])**	It executes the callback function once.
6.	**Set.prototype.has(value)**	This method returns true when the passed value is in the Set.
7.	**Set.prototype.values()**	It returns the new iterator object, which contains the values for each element in the Set, in insertion order.

ES6 Set

Set.prototype.entries() - It returns the object of a new set iterator. It contains an array of values for each element. It maintains the insertion order.

```
let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
colors.add('Violet');
colors.add('Indigo');
colors.add('Blue');
colors.add('Violet');
var itr = colors.entries();
for(i=0;i<colors.size;i++)  {
   console.log(itr.next().value);
}
```

Set.prototype.forEach(callbackFn[, thisArg]) - It executes the specified callback function once for each Map entry.

```
let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
colors.add('Violet');
colors.add('Indigo');
colors.add('Blue');
colors.add('Violet');
function details(values){
   console.log(values);
}
colors.forEach(details);
```

ES6 Set

Set.prototype.has(value) - It returns the Boolean value that indicates whether the element, along with the corresponding value, exists in a Set object or not.

```
let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
colors.add('Violet');
colors.add('Indigo');
colors.add('Blue');
colors.add('Violet');
console.log(colors.has('Indigo'));
console.log(colors.has('Violet'));
console.log(colors.has('Cyan'));
```

Set.prototype.values() -It returns a new iterator object that includes the values for each element in the Set object in the insertion order.

```
let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
colors.add('Violet');

var val = colors.values();
console.log(val.next().value);
console.log(val.next().value);
console.log(val.next().value);
console.log(val.next().value);
console.log(val.next().value);
```

ES6 Set

Weak Set –

It is used to store the collection of objects. It is similar to the Set object, so it also cannot store duplicate values. Similar to weak maps, weak sets cannot be iterated. Weak sets can contain only objects which may be garbage collected.

Weak set only includes **add(value), delete(value)** and **has(value)** methods of Set object.

```
'use strict'
  let ws = new WeakSet();
  let obj = {msg:"Welcome Back!"};
  ws.add(obj);
  console.log(ws.has(obj));
  ws.delete(obj);
  console.log(ws.has(obj));
```

Iterators -An iterator is an object which defines the sequence and a return value upon its termination. It allows accessing a collection of objects one at a time. Set and Map both include the methods that return an iterator.

Iterators are the objects with the next() method. When the next() method gets invoked, the iterator returns an object along with the 'value' and 'done' properties.

The 'done' is a Boolean which returns true after reading all of the elements in the collection. Otherwise, it returns false.

```
let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
var itr = colors.keys();
var itr1 = colors.entries();
var itr2 = colors.values();
console.log(itr.next());
console.log(itr1.next());
console.log(itr2.next());
```

Iterators

The for loop issues

When you have an array of data, you typically use a for loop to iterate over its elements.

```
let ranks = ['A', 'B', 'C'];

for (let i = 0; i < ranks.length; i++) {
   console.log(ranks[i]);
}
```

The for loop uses the variable i to track the index of the ranks array. The value of i increments each time the loop executes as long as the value of i is less than the number of elements in the ranks array.

This code is straightforward. However, its complexity grows when you nest a loop inside another loop. In addition, keeping track of multiple variables inside the loops is error-prone.

ES6 introduced a new loop construct called for...of to eliminate the standard loop’s complexity and avoid the errors caused by keeping track of loop indexes.

To iterate over the elements of the ranks array, you use the following for...of construct:

```
for(let rank of ranks) {
   console.log(rank);
}
```

The for...of is far more elegant than the for loop because it shows the true intent of the code – iterate over an array to access each element in the sequence.

On top of this, the for...of loop has the ability to create a loop over any iterable object, not just an array.

To understand the iterable object, you need to understand the iteration protocols first.

Iterators

Iteration protocols

There are two iteration protocols: iterable protocol and iterator protocol.

Iterator protocol

An object is an iterator when it implements an interface (or API) that answers two questions:

Is there any element left?

If there is, what is the element?

Technically speaking, an object is qualified as an iterator when it has a next() method that returns an object with two properties:

done: a Boolean value indicating whether or not there are any more elements that could be iterated upon.

value: the current element.

Each time you call the next(), it returns the next value in the collection:

```
{ value: 'next value', done: false }
```

If you call the next() method after the last value has been returned, the next() returns the result object as follows:

```
{done: true: value: undefined}
```

The value of the done property indicates that there is no more value to return and the value of the property is set to undefined.

Iterable protocol

An object is iterable when it contains a method called [Symbol.iterator] that takes no argument and returns an object which conforms to the iterator protocol.

The [Symbol.iterator] is one of the built-in well-known symbols in ES6.

Iterators

Iterators

Since ES6 provides built-in iterators for the collection types Array, Set, and Map, you don’t have to create iterators for these objects.

If you have a custom type and want to make it iterable so that you can use the for...of loop construct, you need to implement the iteration protocols.

The following code creates a Sequence object that returns a list of numbers in the range of (start, end) with an interval between subsequent numbers.

```
class Sequence {
  constructor( start = 0, end = Infinity, interval = 1 ) {
    this.start = start;
    this.end = end;
    this.interval = interval;
  }
  [Symbol.iterator]() {
    let counter = 0;
    let nextIndex = this.start;
    return  {
      next: () => {
        if ( nextIndex <= this.end ) {
          let result = { value: nextIndex,  done: false }
          nextIndex += this.interval;
          counter++;
          return result;
        }
        return { value: counter, done: true };
      }
    }
  }
};
```

Iterators

Iterators

The following code uses the Sequence iterator in a for...of loop:

```
let evenNumbers = new Sequence(2, 10, 2);

for (const num of evenNumbers) {
   console.log(num);
}  // 2 4 6 8 10
```

You can explicitly access the [Symbol.iterator]() method as shown in the following script:

```
let evenNumbers = new Sequence(2, 10, 2);
let iterator = evenNumbers[Symbol.iterator]();

let result = iterator.next();

while( !result.done ) {
   console.log(result.value);
   result = iterator.next();
}
```

Iterators

Cleaning up

In addition to the next() method, the [Symbol.iterator]() may optionally return a method called return().

The return() method is invoked automatically when the iteration is stopped prematurely. It is where you can place the code to clean up the resources.

The following example implements the return() method for the Sequence object:

```
class Sequence {
    constructor( start = 0, end = Infinity, interval = 1 ) {
        this.start = start;
        this.end = end;
        this.interval = interval;
    }
    [Symbol.iterator]() {
        let counter = 0;
        let nextIndex = this.start;
        return  {
            next: () => {
                if ( nextIndex <= this.end ) {
                    let result = { value: nextIndex,  done: false }
                    nextIndex += this.interval;
                    counter++;
                    return result;
                }
                return { value: counter, done: true };
            },
            return: () => {
                console.log('cleaning up...');
                return { value: undefined, done: true };
            }
        }
    }
}
```

Iterators

Cleaning up

The following snippet uses the Sequence object to generate a sequence of odd numbers from 1 to 10. However, it prematurely stops the iteration. As a result, the return() method is automatically invoked.

```
let oddNumbers = new Sequence(1, 10, 2);

for (const num of oddNumbers) {
   if( num > 7 ) {
      break;
   }
   console.log(num);
}
```

Output:

```
1
3
5
7
cleaning up...
```

Generators

In JavaScript, a regular function is executed based on the run-to-completion model. It cannot pause midway and then continues from where it paused. For example:

```
function foo() {
   console.log('I');
    //stop 2 seconds
   console.log('cannot');
   console.log('pause');
}
```

The foo() function executes from top to bottom. The only way to exit the foo() is by returning from it or throwing an error. If you invoke the foo() function again, it will start the execution from the top to bottom.

ES6 introduces a new kind of function that is different from a regular function: function generator or generator.
A generator can pause midway and then continues from where it paused. For example:

```
function* generate() {
   console.log('invoked 1st time');
   yield 1;
   console.log('invoked 2nd time');
   yield 2;
}
```

Let’s examine the generate() function in detail.
First, you see the asterisk (*) after the function keyword. The asterisk denotes that the generate() is a generator, not a normal function.
Second, the yield statement returns a value and pauses the execution of the function.

Generators

The following code invokes the generate() generator:

```
let gen = generate();
```

When you invoke the generate() generator:
First, you see nothing in the console. If the generate() were a regular function, you would expect to see some messages.
Second, you get something back from generate() as a returned value.

```
function* generate() {
   console.log('invoked 1st time');
   yield 1;
   console.log('invoked 2nd time');
   yield 2;
}

console.log(gen);
Output: Object [Generator] {}
```

So, a generator returns a Generator object without executing its body when it is invoked.
The Generator object returns another object with two properties: done and value. In other words, a Generator object is **iterable**.
The following calls the next() method on the Generator object:

```
let result = gen.next();
console.log(result);

Output: invoked 1st time
{ value: 1, done: false }
```

Generators

As you can see, the Generator object executes its body which outputs message 'invoked 1st time' at line 1 and returns the value 1 at line 2.
The yield statement returns 1 and pauses the generator at line 2.
Similarly, the following code invokes the next() method of the Generator second time:

```
result = gen.next();
console.log(result);
```

```
Output: invoked 2nd time
{ value: 2, done: false }
```

This time the Generator resumes its execution from line 3 that outputs the message 'invoked 2nd time' and returns (or yield) 2.
The following invokes the next() method of the generator object third time:

```
result = gen.next();
console.log(result);
```

```
Output: { value: undefined, done: true }
```

Since a generator is iterable, you can use the for...of loop:

```
for (const g of gen) {
   console.log(g);
}
```

```
Output: invoked 1st time
1
invoked 2nd time
2
```

Generators

More generator examples

use a generator to generate a never-ending sequence

```
function* forever() {
    let index = 0;
    while (true) {
        yield index++;
    }
}

let f = forever();
console.log(f.next()); // 0
console.log(f.next()); // 1
console.log(f.next()); // 2
```

Each time you call the next() method of the forever generator, it returns the next number in the sequence starting from 0.

Generators

Using generators to implement iterators

When you implement an iterator, you have to manually define the next() method. In the next() method, you also have to manually save the state of the current element.

Since generators are iterables, they can help you simplify the code for implementing iterator.

The following is a Sequence iterator created in the iterator tutorial:

```
class Sequence {
  constructor( start = 0, end = Infinity, interval = 1 ) {
    this.start = start;
    this.end = end;
    this.interval = interval;
  }
  [Symbol.iterator]() {
    let counter = 0;
    let nextIndex = this.start;
    return  {
      next: () => {
        if ( nextIndex < this.end ) {
          let result = { value: nextIndex,  done: false }
          nextIndex += this.interval;
          counter++;
          return result;
        }
        return { value: counter, done: true };
      }
    }
  }
}
```

Generators

Using generators to implement iterators
And here is the new Sequence iterator that uses a generator

```
class Sequence {
  constructor( start = 0, end = Infinity, interval = 1 ) {
    this.start = start;
    this.end = end;
    this.interval = interval;
  }
  * [Symbol.iterator]() {
    for( let index = this.start; index <= this.end; index += this.interval ) {
      yield index;
    }
  }
}
```

As you an see, the method Symbol.iterator is much simpler by using the generator.

The following script uses the Sequence iterator to generate a sequence of odd numbers from 1 to 10:
let oddNumbers = new Sequence(1, 10, 2);

```
for (const num of oddNumbers) {
  console.log(num);
}
Output: 1 3 5 7 9
```

Generators

Using a generator to implement the Bag data structure

A Bag is a data structure that has the ability to collect elements and iterate through elements. It doesn’t support removing items.

```
class Bag {
   constructor() {
      this.elements = [];
   }
   isEmpty() {
      return this.elements.length === 0;
   }
   add(element) {
      this.elements.push(element);
   }
   * [Symbol.iterator]() {
      for (let element of this.elements) {
         yield element;
      }
   }
}
let bag = new Bag();
bag.add(1);
bag.add(2);
bag.add(3);

for (let e of bag) {
   console.log(e);
}
Output: 1 2 3
```

Generators

Summary:

- Generators are created by the generator function function* f(){}.
- Generators do not execute its body immediately when they are invoked.
- Generators can pause midway and resumes their executions where they were paused. The yield statement pauses the execution of a generator and returns a value.
- Generators are iterable so you can use them with the for...of loop.

Yield keyword

Introduction:
The yield keyword allows you to pause and resume a generator function (function*).
The following shows the syntax of the yield keyword:

```
[variable_name] = yield [expression];
```

In this syntax:

- The expression specifies the value to return from a generator function via the iteration protocol. If you omit the expression, the yield returns undefined.
- The variable_name stores the optional value passed to the next() method of the iterator object.

JavaScript yield examples
Let’s take some examples of using the yield keyword.

A) Returning a value
The following trivial example illustrates how to use the yield keyword to return a value from a generator function:

```
function* foo() {
   yield 1;
   yield 2;
   yield 3;
}
let f = foo();
console.log(f.next());
Output:  { value: 1, done: false }
```

As you can see the value that follows the yield is added to the value property of the return object when the next() is called:
yield 1;

Yield keyword

B) Returning undefined

This example illustrates how to use the yield keyword to return undefined:

```
function* bar() {
   yield;
}

let b = bar();
console.log(b.next());
```

Output:

```
{ value: undefined, done: false }
```

Yield keyword

C) Passing a value to the next() method

In the following example, the yield keyword is an expression that evaluates to the argument passed to the next() method:

```
function* generate() {
   let result = yield;
   console.log(`result is ${result}`);
}

let g = generate();
console.log(g.next());

console.log(g.next(1000));
```

The first call g.next() returns the following object:

```
{ value: undefined, done: false }
```

The second call g.next() carries the following tasks:

- Evaluate yield to 1000.
- Assign result the value of yield, which is 1000.
- Output the message and return the object

Output:

```
result is 1000
{ value: undefined, done: true }
```

Yield keyword

D) Using yield in an array

The following example uses the yield keyword as elements of an array:

```
function* baz() {
   let arr = [yield, yield];
   console.log(arr);
}

var z = baz();

console.log(z.next());
console.log(z.next(1));
console.log(z.next(2));
```

The first call z.next() sets the first element of the arr array to 1 and returns the following object:

```
{ value: undefined, done: false }
```

The second call z.next() sets the second of the arr array to 2 and returns the following object:

```
{ value: undefined, done: false }
```

The third call z.next() shows the contents of the arr array and returns the following object:

```
[ 1, 2 ]
{ value: undefined, done: true }
```

Yield keyword

E) Using yield to return an array

The following generator function uses the yield keyword to return an array:

```
function* yieldArray() {
   yield 1;
   yield [ 20, 30, 40 ];
}

let y = yieldArray();

console.log(y.next());
console.log(y.next());
console.log(y.next());
```

The first call y.next() returns the following object:

```
{ value: 1, done: false }
```

The second call y.next() returns the following object:

```
{ value: [ 20, 30, 40 ], done: false }
```

In this case, yield sets the array [20, 30, 40] as the value of the value property of the return object.

The third call y.next() returns the following object:

```
{ value: undefined, done: true }
```

Yield keyword

F) Using the yield to return individual elements of an array

See the following generator function:

```
function* yieldArrayElements() {
   yield 1;
   yield* [ 20, 30, 40 ];
}

let a = yieldArrayElements();

console.log(a.next()); // { value: 1, done: false }
console.log(a.next()); // { value: 20, done: false }
console.log(a.next()); // { value: 30, done: false }
console.log(a.next()); // { value: 40, done: false }
```

In this example, yield* is the new syntax. The yield* expression is used to delegate to another iterable object or generator.

As a result, the following expression returns the individual elements of the array [20, 30, 40]:

```
yield* [20, 30, 40];
```

Yield keyword

G) Once returned, no more yields

See the following generator function:

```
function* yieldAndReturn() {
   yield "Y";
   return "R";
   yield "unreachable";
   }

   var gen = yieldAndReturn()
   console.log(gen.next());
   console.log(gen.next());
   console.log(gen.next());
```

In this example, wonder what would happen if you execute a return statement inside a generator? Well, I have demonstrated just that in the below example. It returns from the generator function without providing access to any of the yield below.

As a result, the output is given below. You cannot yield the "unreachable“

```
{"value":"Y","done":false}
{"value":"R","done":true}
{"done":true}
```

Yield keyword

Advantages of using Generators

1. Lazy Evaluation - Run only when you need

Say there is an Infinite Stream of data, we cannot spend our whole life evaluating that data. Hence we can use Generator function to evaluate as and when required.

2. Memory Efficient

As the Lazy Evaluation method is used, only those data and those computations that are necessary, are used.

Modules

Modules are the piece or chunk of a JavaScript code written in a file. JavaScript modules help us to modularize the code simply by partitioning the entire code into modules that can be imported from anywhere. Modules make it easy to maintain the code, debug the code, and reuse the piece of code. Each module is a piece of code that gets executed once it is loaded.

Modules in ES6 is an essential concept. Although it is not available everywhere, but today we can use it and can transpile into ES5 code. Transpilation is the process of converting the code from one language into its equivalent language. The ES6 module transpiler tool is responsible for taking the ES6 module and converts it into a compatible code of ES5 in the AMD **(Asynchronous module definition** is a specification for the JavaScript programming language**)** or in the CommonJS style.

During the build process, we can use **Gulp, Babel, Grunt,** or other **transpilers** for compiling the modules. The variables and functions in a module are not available for use unless the file exports them.

Modules

Exporting and Importing a Module

Exporting a Module

JavaScript allows us to export a function, objects, classes, and primitive values by using the **export** keyword. There are two kinds of exports:

- Named Exports: The exports that are distinguished with their names are called as named exports. We can export multiple variables and functions by using the named export.
- Default Exports: There can be, at most, one value that can be exported by using the default export.

Importing a Module

To import a module, we need to use the import keyword. The values which are exported from the module can be imported by using the import keyword. We can import the exported variables, functions, and classes in another module. To import a module, we simply have to specify their path.

When you import the named export, you must have to use the same name as the corresponding object. When you import the default export, we can use any name of the corresponding object.

Modules

Named Exports and Imports

Named exports are distinguished with their names. The class, variable, or any function which is exported by using the **named export** can only be imported by using the same name.

Multiple variables and functions can be imported and exported by using the **named** export.

```
//Named export in class
class Nokia{
//properties
//methods
}
export {Nokia}; //Named export
//Named export in functions
function show(){
}
export {show};
//Named export in Variables
const a = 10;
export {a};
```

Modules

We can apply more than one named export in a module. We can use the syntax of multiple named exports in a module as follows:

```
class Nokia{
//properties
//methods
}
function show(){
}
const a = 10;
export {Nokia, show};
```

Modules

Importing the Named export

To import the bindings that are exported by another module, we have to use the static import statement. The imported modules are always in the **strict mode**, whether we declare them in strict mode or not.

import {Nokia, show} from './Mobile.js’;

Importing all

If we want to import all of the export statements simultaneously, then we can import them individually.

But it will be hard when we have so many named exports. So, to make it easier, we can do it as follows:

import * as device from './Mobile.js'; // Here, the device is an alias, and Mobile.js is the module name.

Suppose, we have defined a class Nokia in the module Mobile.js, and if we want to use it then by using the alias, we can perform it as:

device.Nokia //if we have a class Nokia

device.show // if we have a function show

device.a // if we have a variable a

Asynchronous

Waiting for a Timeout

When using the JavaScript function setTimeout(), you can specify a callback function to be executed on time-out

```
setTimeout(myFunction, 3000);

function myFunction() {
  document.getElementById("demo").innerHTML = "I can’t find You !!";
}
```

In the example above, myFunction is used as a callback.

The function (the function name) is passed to setTimeout() as an argument.

3000 is the number of milliseconds before time-out, so myFunction() will be called after 3 seconds.

Waiting for Intervals:

```
setInterval(myFunction, 1000);

function myFunction() {
  let d = new Date();
  document.getElementById("demo").innerHTML=
  d.getHours() + ":" +
  d.getMinutes() + ":" +
  d.getSeconds();
}
```

Asynchronous

Waiting for Files

If you create a function to load an external resource (like a script or a file), you cannot use the content before it is fully loaded.

This is the perfect time to use a callback.

This example loads a HTML file (mycar.html), and displays the HTML file in a web page, after the file is fully loaded:

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}

function getFile(myCallback) {
  let req = new XMLHttpRequest();
  req.open('GET', "mycar.html");
  req.onload = function() {
    if (req.status == 200) {
      myCallback(this.responseText);
    } else {
      myCallback("Error: " + req.status);
    }
  }
  req.send();
}

getFile(myDisplayer)
```

In the example above, myDisplayer is used as a callback.

The function (the function name) is passed to getFile() as an argument.

Promises

A JavaScript Promise object contains both the producing code and calls to the consuming code

```
let myPromise = new Promise(function(myResolve, myReject) {
// "Producing Code" (May take some time)

  myResolve(); // when successful
  myReject();  // when error
});

// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

When the executing code obtains the result, it should call one of the two callbacks:

Result	Call
Success	myResolve(result value)
Error	myReject(error object)

Promise Object Properties

A JavaScript Promise object can be:

- Pending
- Fulfilled
- Rejected

The Promise object supports two properties: **state** and **result**.
While a Promise object is "pending" (working), the result is undefined.
When a Promise object is "fulfilled", the result is a value.
When a Promise object is "rejected", the result is an error object.

myPromise.state	**myPromise.result**
"pending"	undefined
"fulfilled"	a result value
"rejected"	an error object

You cannot access the Promise properties **state** and **result**.
You must use a Promise method to handle promises.

Promises

Promise Example:

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}

let myPromise = new Promise(function(myResolve, myReject) {
  let x = 0;

// The producing code (this may take some time)

  if (x == 0) {
    myResolve("OK");
  } else {
    myReject("Error");
  }
});

myPromise.then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);
```

Promises

Waiting for a file Example:

```
let myPromise = new Promise(function(myResolve, myReject) {
  let req = new XMLHttpRequest();
  req.open('GET', "mycar.htm");
  req.onload = function() {
    if (req.status == 200) {
      myResolve(req.response);
    } else {
      myReject("File not Found");
    }
  };
  req.send();
});

myPromise.then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);
```

Promises

Example

The example given below shows a function add_positivenos_async() which adds two numbers asynchronously. The promise is resolved if positive values are passed. The promise is rejected if negative values are passed.

```
<script>
  function add_positivenos_async(n1, n2) {
    let p = new Promise(function (resolve, reject) {
      if (n1 >= 0 && n2 >= 0) {
        //do some complex time consuming work
        resolve(n1 + n2)
      }
      else
        reject('NOT_Postive_Number_Passed')
      })
      return p;
  }
  add_positivenos_async(10, 20)
    .then(successHandler) // if promise resolved
    .catch(errorHandler);// if promise rejected
  add_positivenos_async(-10, -20)
    .then(successHandler) // if promise resolved
    .catch(errorHandler);// if promise rejected
  function errorHandler(err) {
    console.log('Handling error', err)
  }
  function successHandler(result) {
    console.log('Handling success', result)
  }
  console.log('end')
</script>
```

The output of the above code will be as mentioned below –

```
end
```

Promises

Promises Chaining

Promises chaining can be used when we have a sequence of asynchronous tasks to be done one after another. Promises are chained when a promise depends on the result of another promise. This is shown in the example below

Example

In the below example, add_positivenos_async() function adds two numbers asynchronously and rejects if negative values are passed. The result from the current asynchronous function call is passed as parameter to the subsequent function calls. Note each then() method has a return statement.

```
<script>
  function add_positivenos_async(n1, n2) {
    let p = new Promise(function (resolve, reject) {
      if (n1 >= 0 && n2 >= 0) {
        //do some complex time consuming work
        resolve(n1 + n2)
      }
      else
        reject('NOT_Postive_Number_Passed')
    })
    return p;
  }

  add_positivenos_async(10,20)
  .then(function(result){
    console.log("first result",result)
    return add_positivenos_async(result,result)
  }).then(function(result){
  console.log("second result",result)
    return add_positivenos_async(result,result)
  }).then(function(result){
    console.log("third result",result)
  })

  console.log('end')
```

Promises

promise.all()
This method can be useful for aggregating the results of multiple promises.

Syntax
The syntax for the promise.all() method is mentioned below, where, iterable is an iterable object. E.g. Array.

Promise.all(iterable);
Example
The example given below executes an array of asynchronous operations [add_positivenos_async(10,20),add_positivenos_async(30,40),add_positivenos_async(50,60)]. When all the operations are completed, the promise is fully resolved.

```
<script>
  function add_positivenos_async(n1, n2) {
    let p = new Promise(function (resolve, reject) {
      if (n1 >= 0 && n2 >= 0) {
        //do some complex time consuming work
        resolve(n1 + n2)
      }
      else
        reject('NOT_Postive_Number_Passed')
    })

    return p;
  }
  //Promise.all(iterable)

Promise.all([add_positivenos_async(10,20),add_positivenos_async(30,40),add_positivenos_async(50,60)])
  .then(function(resolveValue){
    console.log(resolveValue[0])
    console.log(resolveValue[1])
    console.log(resolveValue[2])
```

Promises

promise.race()
This function takes an array of promises and returns the first promise that is settled.

Syntax
The syntax for the promise.race() function is mentioned below, where, iterable is an iterable object. E.g. Array.

Promise.race(iterable)
Example
The example given below takes an array [add_positivenos_async(10,20),add_positivenos_async(30,40)] of asynchronous operations.

The promise is resolved whenever any one of the add operation completes. The promise will not wait for other asynchronous operations to complete.

```
<script>
  function add_positivenos_async(n1, n2) {
    let p = new Promise(function (resolve, reject) {
      if (n1 >= 0 && n2 >= 0) {
        //do some complex time consuming work
        resolve(n1 + n2)
      } else
        reject('NOT_Postive_Number_Passed')
    })

    return p;
  }

  //Promise.race(iterable)
  Promise.race([add_positivenos_async(10,20),add_positivenos_async(30,40)])
  .then(function(resolveValue){
    console.log('one of them is done')
    console.log(resolveValue)
  }).catch(function(err){
```

Promises

Promises are a clean way to implement async programming in JavaScript (ES6 new feature). Prior to promises, Callbacks were used to implement async programming. Let’s begin by understanding what async programming is and its implementation, using Callbacks.

Understanding Callback
A function may be passed as a parameter to another function. This mechanism is termed as a Callback. A Callback would be helpful in events.

The following example will help us better understand this concept.

```
<script>
  function notifyAll(fnSms, fnEmail) {
    console.log('starting notification process');
    fnSms();
    fnEmail();
  }
  notifyAll(function() {
    console.log("Sms send ..");
  },
  function() {
    console.log("email send ..");
  });
  console.log("End of script");
  //executes last or blocked by other methods
</script>
```

In the notifyAll() method shown above, the notification happens by sending SMS and by sending an e-mail. Hence, the invoker of the notifyAll method has to pass two functions as parameters. Each function takes up a single responsibility like sending SMS and sending an e-mail.

The following output is displayed on successful execution of the above code.

```
starting notification process
Sms send ..
Email send ..
```

Promises

Understanding AsyncCallback
Consider the above example.

To enable the script, execute an asynchronous or a non-blocking call to notifyAll() method. We shall use the setTimeout() method of JavaScript. This method is async by default.

The setTimeout() method takes two parameters −

A callback function.

The number of seconds after which the method will be called.

In this case, the notification process has been wrapped with timeout. Hence, it will take a two seconds delay, set by the code. The notifyAll() will be invoked and the main thread goes ahead like executing other methods. Hence, the notification process will not block the main JavaScript thread.

```
<script>
  function notifyAll(fnSms, fnEmail) {
    setTimeout(function() {
      console.log('starting notification process');
      fnSms();
      fnEmail();
    }, 2000);
  }
  notifyAll(function() {
    console.log("Sms send ..");
  },
  function() {
    console.log("email send ..");
  });
  console.log("End of script"); //executes first or not blocked by others
</script>
```

The following output is displayed on successful execution of the above code.

Promises

In case of multiple callbacks, the code will look **scary**.

```
<script>
   setTimeout(function() {
      console.log("one");
      setTimeout(function() {
         console.log("two");
         setTimeout(function() {
            console.log("three");
         }, 1000);
      }, 1000);
   }, 1000);
</script>
```

ES6 comes to your rescue by introducing the concept of promises. Promises are "Continuation events" and they help you execute the multiple async operations together in a much cleaner code style.

Example
Let's understand this with an example. Following is the syntax for the same.

```
var promise = new Promise(function(resolve , reject) {
   // do a thing, possibly async , then..
   if(/*everthing turned out fine */)    resolve("stuff worked");
   else
   reject(Error("It broke"));
});
return promise;
// Give this to someone
```

Promises

The first step towards implementing the promises is to create a method which will use the promise. Let’s say in this example, the getSum() method is asynchronous i.e., its operation should not block other methods’ execution. As soon as this operation completes, it will later notify the caller.

The following example (Step 1) declares a Promise object ‘var promise’. The Promise Constructor takes to the functions first for the successful completion of the work and another in case an error happens.

The promise returns the result of the calculation by using the resolve callback and passing in the result, i.e., n1+n2

Step 1 − resolve(n1 + n2);

If the getSum() encounters an error or an unexpected condition, it will invoke the reject callback method in the Promise and pass the error information to the caller.

Step 2 − reject(Error("Negatives not supported"));

The method implementation is given in the following code (STEP 1).

```
function getSum(n1, n2) {
  varisAnyNegative = function() {
    return n1 < 0 || n2 < 0;
  }
  var promise = new Promise(function(resolve, reject) {
    if (isAnyNegative()) {
      reject(Error("Negatives not supported"));
    }
    resolve(n1 + n2)
  });
  return promise;
}
```

Promises

The second step details the implementation of the caller (STEP 2).

The caller should use the ‘then’ method, which takes two callback methods - first for success and second for failure. Each method takes one parameter, as shown in the following code.

```
getSum(5, 6)
.then(function (result) {
  console.log(result);
},
function (error) {
  console.log(error);
});
```

The following output is displayed on successful execution of the above code.

```
11
```

Since the return type of the getSum() is a Promise, we can actually have multiple ‘then’ statements. The first 'then' will have a return statement.

```
getSum(5, 6)
.then(function(result) {
  console.log(result);
  returngetSum(10, 20);
  // this returns another promise
},
function(error) {
  console.log(error);
})
.then(function(result) {
  console.log(result);
},
function(error) {
  console.log(error);
});
```

Promises

The following example issues three then() calls with getSum() method.

```
<script>
  function getSum(n1, n2) {
    varisAnyNegative = function() {
      return n1 < 0 || n2 < 0;
    }
    var promise = new Promise(function(resolve, reject) {
      if (isAnyNegative()) {
        reject(Error("Negatives not supported"));
      }
      resolve(n1 + n2);
    });
    return promise;
  }
  getSum(5, 6)
  .then(function(result) {
    console.log(result);
    returngetSum(10, 20);
    //this returns another Promise
  },
  function(error) {
    console.log(error);
  })
  .then(function(result) {
    console.log(result);
    returngetSum(30, 40);
    //this returns another Promise
  },
  function(error) {
    console.log(error);
```

Promises

The following output is displayed on successful execution of the above code.

The program displays ‘end of script’ first and then results from calling getSum() method, one by one.

```
End of script
11
30
70
```

This shows getSum() is called in async style or non-blocking style. Promise gives a nice and clean way to deal with the Callbacks

Async-await

Async and await make promises easier to write, async makes a function return a Promise, await makes a function wait for a Promise

The keyword async before a function makes the function return a promise

```
async function myFunction() {
  return "Hello";
}

Same as:

async function myFunction() {
  return Promise.resolve("Hello");
}

//Then

async function myFunction() {
  return "Hello";
}
myFunction().then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);

//Or simpler, since you expect a normal value (a normal response, not an error):
async function myFunction() {
  return "Hello";
}
myFunction().then(
  function(value) {myDisplayer(value);}
);
```

Async-await

Await Syntax

The keyword await before a function makes the function wait for a promise:

```
let value = await promise;
```

The await keyword can only be used inside an async function.

Example

Let's go slowly and learn how to use it.

Basic Syntax

```
async function myDisplay() {
  let myPromise = new Promise(function(myResolve, myReject) {
    myResolve("I will catch you!");
  });
  document.getElementById("demo").innerHTML = await myPromise;
}
myDisplay();
```

Async-await

Waiting for a Timeout

```
async function myDisplay() {
  let myPromise = new Promise(function(myResolve, myReject) {
    setTimeout(function() { myResolve("I will find you!!"); }, 3000);
  });
  document.getElementById("demo").innerHTML = await myPromise;
}

myDisplay();
```

Async-await

Waiting for a File

```
async function getFile() {
  let myPromise = new Promise(function(myResolve, myReject) {
    let req = new XMLHttpRequest();
    req.open('GET', "mycar.html");
    req.onload = function() {
      if (req.status == 200) {myResolve(req.response);}
      else {myResolve("File not Found");}
    };
    req.send();
  });
  document.getElementById("demo").innerHTML = await myPromise;
}

getFile();
```