

Java Script

Java script is a scripting language for web application development. It is the foundation of frontend web development and is the key ingredient in frameworks like ReactJS, Angular, and VueJS.

It can also help create solid backends with platforms like Nodejs, runs desktop applications like Slack, Atom, and Spotify, and runs on mobile phones as Progressive Web Apps (PWAs).

JavaScript is a lightweight, cross-platform, and interpreted compiled programming language which is also known as the scripting language for webpages.

JavaScript can be used for **Client-side** developments as well as **Server-side** developments.

Java script is both imperative and declarative type of language. (It means Declarative programming tells the computer "What" the program should do, Imperative programming tells the machine "How" to do it.

JavaScript is *an object-based scripting language*.

Application of JavaScript

JavaScript is used to create interactive websites. It is mainly used for:

- Client-side validation,
- Dynamic drop-down menus,
- Displaying date and time,
- Displaying pop-up windows and dialog boxes (like an alert dialog box, confirm dialog box and prompt dialog box),
- Displaying clocks etc.
- Games, Smart watches, Web Application Design, Mobile App Design

JAVA SCRIPT FEATURES:

1. High Level
2. Dynamic
3. Dynamically typed
4. Loosely typed
5. Interpreted
6. Multi-paradigm

JavaScript types

The set of types in the JavaScript language consists of *primitive values* and *objects*.

- 1) Primitive values (immutable datum represented directly at the lowest level of the language)
 - Boolean type
 - Null type
 - Undefined type
 - Number type
 - BigInt type (it is created by add 'n' at the end)
 - String type
 - Symbol type
- 2) Objects (collections of properties)
 - Array
 - Date
 - JSON

Null type: The Null type is inhabited by exactly one value: null.

Undefined type: The Undefined type is inhabited by exactly one value: undefined. Conceptually, undefined indicates the absence of a value, while null indicates the absence of an object.

Boolean type: The Boolean type represents a logical entity and is inhabited by two values: true and false.

Number type: The Number type is a double-precision 64-bit binary format IEEE 754 value. It is capable of storing positive floating-point numbers between 2×10^{74} (Number.MIN_VALUE) and 2^{1024} (Number.MAX_VALUE) as well as negative floating-point numbers between -2×10^{74} and -2^{1024} , but it can only safely store integers in the range $-(2^{53} - 1)$ (Number.MIN_SAFE_INTEGER) to $2^{53} - 1$

BigInt type: The BigInt type is a numeric primitive in JavaScript that can represent integers with arbitrary magnitude.

ex: `const x = BigInt(Number.MAX_SAFE_INTEGER); // 9007199254740991n`

String type: The String type represents textual data and is encoded as a sequence of 16-bit unsigned integer values representing UTF-16 code units. Each element in the string occupies a position in the string. The first element is at index 0, the next at index 1, and so on. JavaScript strings are immutable. This means that once a string is created, it is not possible to modify it.

Symbol type: A Symbol is a unique and immutable primitive value and may be used as the key of an Object property.

Objects: A JavaScript object is a mapping between *keys* and *values*. Keys are strings (or Symbols), and *values* can be anything. This makes objects a natural fit for hashmaps. Functions are regular objects with the additional capability of being *callable*. Objects can be represented in `{ }`.

An object can be – Array, Date, JSON

Ex-1:

```
var object1= {name:"Varun", age:20};
var object1= {name:"Vishnu", age:23};
console.log(object1);
console.log(object2);
```

Ex-2:

```
var obj1 = { a: 5, b: 6 };
obj1[a]=7;
console.log(obj1) // will return the value {a: 7, b: 6}
```

Objects can be handled in 3 different types of objects: individual objects, nested objects and array of objects.

Ex:

```
let person = {                                // individual object
  firstName: 'Ravi',
  lastName: 'xyz',
  age: 20,
  gender: 'male'
};
```

```
let person = {                                // nested object
  firstName: 'Ravi',
  lastName: 'xyz',
  address: {
    place: '2/12 Brodipet',
    city: 'Guntur',
  }
};
```

```
        state: 'Andhra Pradesh',  
        country: 'India' }  
};  
  
let products = [                // array of objects  
    {name: 'iPhone', price: 900},  
    {name: 'Samsung Galaxy', price: 850},  
    {name: 'Sony Xperia', price: 700}  
];
```

A Java script program is a functional programming language. It doesn't have any specific program structure. A java script program is embedded in an HTML program through `<script>` `</script>` tags.

Ex: A simple java script program.

```
<!DOCTYPE html>  
<html lang="en">  
  
<head>  
  <title> Example Demo for Java script </title>  
</head>  
<body>  
  <script>  
    console.log ("Welcome to Java script");  
  </script>  
</body>  
</html>
```

JavaScript Output Statements: JavaScript can "display" data in different ways:

- Writing into an HTML element, using innerHTML.
- Writing into the HTML output using document.write().
- Writing into an alert box, using window.alert().
- Writing into the browser console, using console.log().

Java script consists of different objects to represent browser, and the HTML document. The objects are given below:

Window Object

- Window object is a top-level object in Client-Side JavaScript.
- Window object represents the browser's window.
- It represents an open window in a browser.
- It supports all browsers.
- The document object is a property of the window object. So, typing window.document.write is same as document.write.
- All global variables are properties and functions are methods of the window object.

In addition to Window, the other objects in Java script are: Document, Location, History, Navigator, Date, Math etc.,

Window object properties & methods are given below:

Property	Description
Document	It returns the document object for the window (DOM).
Frames	It returns an array of all the frames including iframes in the current window.
Closed	It returns the boolean value indicating whether a window has been closed or not.
History	It returns the history object for the window.
innerHeight	It sets or returns the inner height of a window's content area.
innerWidth	It sets or returns the inner width of a window's content area.
Length	It returns the number of frames in a window.
Location	It returns the location object for the window.
Name	It sets or returns the name of a window.
Navigator	It returns the navigator object for the window.
outerHeight	It sets or returns the outer height of a window, including toolbars/scrollbars.
outerWidth	It sets or returns the outer width of a window, including toolbars/scrollbars.
Parent	It returns the parent window of the current window.
Screen	It returns the screen object for the window.
screenX	It returns the X coordinate of the window relative to the screen.
screenY	It returns the Y coordinate of the window relative to the screen.
Self	It returns the current window.
Status	It sets the text in the status bar of a window.

Method	Description
alert()	It displays an alert box.
confirm()	It displays a dialog box.
prompt()	It displays a dialog box that prompts the visitor for input.
setInterval()	It calls a function or evaluates an expression at specified intervals.
setTimeout()	It evaluates an expression after a specified number of milliseconds.
clearInterval()	It clears a timer specified by setInterval().

clearTimeout()	It clears a timer specified by setTimeout().
close()	It closes the current window.
open()	It opens a new browser window.
createPopup()	It creates a pop-up window.
focus()	It sets focus to the current window.
blur()	It removes focus from the current window.
moveBy()	It moves a window relative to its current position.
moveTo()	It moves a window to the specified position.
resizeBy()	It resizes the window by the specified pixels.
resizeTo()	It resizes the window to the specified width and height.
print()	It prints the content of the current window.
scrollBy()	It scrolls the content by the specified number of pixels.
scrollTo()	It scrolls the content to the specified coordinates.

History Object

- History object is a part of the window object.
- It is accessed through the window.history property.
- It contains the information of the URLs visited by the user within a browser window.

Method	Description
back()	It loads the previous URL in the history list.
forward()	It loads the next URL in the history list.
go("URL")	It loads a specific URL from the history list.

Location Object

Location object is a part of the window object. It is accessed through the '**window.location**' property. It contains the information about the current URL.

Method	Description
assign()	It loads a new document.
reload()	It reloads the current document.
replace()	It replaces the current document with a new one.

Event Handling:

Events are actions or occurrences that happen in the system you are programming — the system produces (or "fires") a signal of some kind when an event occurs, and provides a mechanism by which an action can be automatically taken (that is, some code running) when the event occurs. Events are fired inside the browser window, and tend to be attached to a specific item that resides in it. This might be a single element, a set of elements, the HTML document loaded in the current tab, or the entire browser window. There are many different types of events that can occur.

For example:

- The user selects, clicks, or hovers the cursor over a certain element.
- The user chooses a key on the keyboard.
- The user resizes or closes the browser window.
- A web page finishes loading.
- A form is submitted.
- A video is played, paused, or ends.
- An error occurs.

To react to an event, you attach an **event handler** to it. This is a block of code (usually a JavaScript function that you as a programmer create) that runs when the event fires. When such a block of code is defined to run in response to an event, we say we are **registering an event handler**. Note: Event handlers are sometimes called **event listeners**.

Syntax:

```
<element event='some JavaScript'>
```

Here is a list of some common HTML events:

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

Ex-1:

```
<!DOCTYPE html>
<html>
<body>
<button onclick="document.getElementById('demo').innerHTML=Date()">
Get Time</button>
<p id="demo"></p>
</body>
</html>
```

Ex-2:

```
<!DOCTYPE html>
```

```
<html>
<body>
<h1>Event Handling - Demo</h1>
<h2>The onmouseover Event</h2>


<input type="button" value="Incr" id="one" onclick="bigImg(this)">
<input type="button" value="Decr" id="two" onclick="normalImg(this)">

<script>
function bigImg(x) {
    x.style.height = "64px";
    x.style.width = "64px";
}

function normalImg(x) {
    x.style.height = "32px";
    x.style.width = "32px";
}
</script>

</body>
</html>
```

Ex-3:

```
<html>
<head><title>String programs</title></head>

<body onload="reverse()">
<script>
    function reverse()
    {
        var r=prompt("ente the number");
        var t=r.split("").reverse().join("");
        document.write(t);
    }
</script>
</body>
</html>
```

Ex-4: Displaying mouse coordinates on mouse move/click

```
<p id= 'output'></p>
document.onmousemove= mouseCoordinates;
var output= document.getElementById('output');

function mouseCoordinates(event){

var xPos= event.clientX;
var yPos= event.clientY;

output.innerHTML= "Coordinate (X) : " + xPos + " " + "pixels <br>Coordinate (Y) : " + yPos + " " + "pixels";

}
```

Event Bubbling:

Event Bubbling is a concept in the DOM (Document Object Model). It happens when an element receives an event, and that event bubbles up (or you can say is transmitted or propagated) to its parent and ancestor elements in the DOM tree until it gets to the root element.

This is the default behaviour of events on elements unless you stop the propagation. The "Event Bubbling" behaviour makes it possible for you to handle an event in a parent element instead of the actual element that received the event.

Ex:

```
<html>
<head>
  <title>Bubbling Event in Javascript</title>
</head>
<body>
<h2>Bubbling Event in Javascript</h2>
<div id="parent">
  <button>
    <h2>Parent</h2>
  </button>
  <button id="child">
    <p>Child</p>
  </button>
</div>
<br>
<script>
  document.getElementById("child").addEventListener("click", function () {
    alert("You clicked the Child element!");
  }, false);

  document.getElementById("parent").addEventListener("click", function () {
    alert("You clicked the parent element!");
  }, false);
</script>
</body>
</html>
```

Event Bubbling is a default behaviour for events. But in some cases, you might want to prevent this. To prevent event bubbling, you use the stopPropagation method of the event object.

Drag n Drop API:

Drag and drop is a very common feature. It is when you "grab" an object and drag it to a different location.

Ex:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Using Drag and Drop</title>
<script>
  function dragStart(e) {
```



```

    // Sets the operation allowed for a drag source
    e.dataTransfer.effectAllowed = "move";

    // Sets the value and type of the dragged data
    e.dataTransfer.setData("Text", e.target.getAttribute("id"));
}

function dragOver(e) {
    // Prevent the browser default handling of the data
    e.preventDefault();
    e.stopPropagation();
}

function drop(e) {
    // Cancel this event for everyone else
    e.stopPropagation();
    e.preventDefault();

    // Retrieve the dragged data by type
    var data = e.dataTransfer.getData("Text");

    // Append image to the drop box
    e.target.appendChild(document.getElementById(data));
}

</script>

<style>
#dropBox {
    width: 300px;
    height: 300px;
    border: 5px dashed gray;
    background: lightyellow;
    text-align: center;
    margin: 20px 0;
    color: orange;
}
#dropBox img {
    margin: 25px;
}
</style>
</head>
<body>
    <h2>Drag and Drop Demo</h2>
    <p>Drag and drop the image into the drop box:</p>
    <div id="dropBox" ondragover="dragOver(event);" ondrop="drop(event);">
    </div>
    
</body>
</html>

```

Canvas:

<canvas> is an HTML element which can be used to draw graphics via scripting (usually JavaScript). This can, for instance, be used to draw graphs, combine photos, or create simple animations.

Ex-1:

```
<html>
<body>
<button onclick= "draw()">
<script>
    function draw () {
        const canvas = document.getElementById("canvas");
        if (canvas.getContext) {
            const ctx = canvas.getContext("2d");

            ctx.fillRect(25, 25, 100, 100);
            ctx.clearRect(45, 45, 60, 60);
            ctx.strokeRect(50, 50, 50, 50);
        }
    }
</script>
</body>
</html>
```

Ex-2: to display a smiley in the screen using canvas.

```
function draw() {
    const canvas = document.getElementById("canvas");

    if (canvas.getContext) {
        const ctx = canvas.getContext("2d");

        ctx.beginPath();
        ctx.arc(75, 75, 50, 0, Math.PI * 2, true); // Outer circle
        ctx.moveTo(110, 75);
        ctx.arc(75, 75, 35, 0, Math.PI, false); // Mouth (clockwise)
        ctx.moveTo(65, 65);
        ctx.arc(60, 65, 5, 0, Math.PI * 2, true); // Left eye
        ctx.moveTo(95, 65);
        ctx.arc(90, 65, 5, 0, Math.PI * 2, true); // Right eye
        ctx.stroke();
    }
}
```

Geolocation API:

The HTML Geolocation API is used to get the geographical position of a user. Since this can compromise privacy, the position is not available unless the user approves it.

```
<!DOCTYPE html>
<html>
<head>
<title>Geolocation API</title>
</head>
<body>
```

```
<h1>Demo on Geolocation</h1>
<h1>Find your Current location</h1>
<button onclick="getlocation()">Click me</button>
<div id="location"></div>

<script>
    var x= document.getElementById("location");
    function getlocation() {
        if(navigator.geolocation)
            navigator.geolocation.getCurrentPosition(showPosition)
        else
            alert ("Sorry! your browser is not supporting");
    }
    function showPosition(position) {
        var x = "Your current location is (" + "Latitude: " + position.coords.latitude +
        ", " + "Longitude: " + position.coords.longitude + ")";
        document.getElementById("location").innerHTML = x;
    }
</script>
</body>
</html>
```

Functions:

Functions are one of the fundamental building blocks in JavaScript. A function in JavaScript is similar to a procedure—a set of statements that performs a task or calculates a value, but for a procedure to qualify as a function, it should take some input and return an output where there is some obvious relationship between the input and the output.

To use a function, you must define it somewhere in the scope from which you wish to call it.

Syntax:

```
Function function_name(arguments) {
    .....// body of the function
    return value;
}
```

Types of functions: there are several types of functions in Java script. The following is the list.

- 1) Normal functions
- 2) Arrow Functions
- 3) Anonymous Functions
- 4) Call back functions
- 5) Higher order functions
- 6) Nested functions
 - Closures
- 1) Recursive Functions

Normal function: any function with name of the function prefixed by function keyword and a set of arguments is referred as normal function.

Ex:

```
function myfun(message) {
    console.log(message);
```

```
}  
  
let result = myfun('Hello');  
console.log('Result:', result);
```

Arrow function: it is a new kind of function definition given in ES6. A function is defined with => (arrow). An **arrow function expression** is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.

Ex:

```
let sum = (a,b) => {  
    return a+b;  
}
```

Anonymous Functions: It is a nameless function. An anonymous function is a function without a name. The following shows how to define an anonymous function:

Ex:

```
(function () {  
    //...  
});  
function (a) {  
    return a + 100;  
}
```

Arrays & their Methods

The **Array** object, as with arrays in other programming languages, enables storing a collection of multiple items under a single variable name, and has members for performing common array operations.

- JavaScript arrays are resizable and can contain a mix of different data types.
- JavaScript arrays are not associative arrays and so, array elements cannot be accessed using strings as indexes, but must be accessed using integers as indexes.
- JavaScript arrays are zero-indexed.
- JavaScript array-copy operations create shallow copies.

Array Methods:

- static methods
- iterative methods

The following are the static methods:

- 1) `Array.from()`: Creates a new Array instance from an array-like object or iterable object.
- 2) `Array.isArray()`: Returns true if the argument is an array, or false otherwise.
- 3) `Array.of()`: Creates a new Array instance with a variable number of arguments, regardless of number or type of the arguments.

Instance Methods:

<code>find()</code>	<code>at()</code>	<code>fill()</code>	<code>forEach()</code>	<code>from()</code>	<code>includes()</code>
<code>join()</code>	<code>map()</code>	<code>pop()</code>	<code>push()</code>	<code>reduce()</code>	<code>reverse()</code>
<code>slice()</code>	<code>sort()</code>	<code>toString()</code>	<code>some()</code>	<code>filter()</code>	<code>concat()</code>

ex-1:

```
const fruits = [];  
fruits.push("banana", "apple", "peach");  
console.log(fruits.length); // 3
```

ex-2:

```
const fruits = ["Apple", "Banana"];  
const fruitsString = fruits.join(", ");  
console.log(fruitsString);
```

ex-3:

```
const fruits = ["Apple", "Banana"];  
fruits.includes("Banana"); // true  
fruits.includes("Cherry"); // false
```

// If `indexOf()` doesn't return -1, the array contains the given item.

```
fruits.indexOf("Banana") !== -1; // true  
fruits.indexOf("Cherry") !== -1; // false
```

ex-4:

```
const fruits = ["Apple", "Banana", "Orange"];  
const removedItem = fruits.pop();  
console.log(fruits);  
// ["Apple", "Banana"]  
console.log(removedItem);
```

ex-5: // removing multiple items

```
const fruits = ["Apple", "Banana", "Strawberry", "Mango", "Cherry"];  
const start = -3;  
const removedItems = fruits.splice(start);  
console.log(fruits);
```

ex-6:

```
const fruits = ["Apple", "Banana"];
const removedItem = fruits.shift();
console.log(fruits); // ["Banana"]
console.log(removedItem);
```

ex-7: This example uses a [for...of](#) loop to iterate over the fruits array, logging each item to the console.

```
const fruits = ["Apple", "Mango", "Cherry"];
for (const fruit of fruits) {
  console.log(fruit);
}
```

Strings:

The **String** object is used to represent and manipulate a sequence of characters. Strings are useful for holding data that can be represented in text form.

Creation of Strings:

```
const string1 = "A string primitive";
const string2 = 'Also a string primitive';
const string3 = `Yet another string primitive`;
```

```
const p = "krishna"; // string literal
```

```
const q = new String("prasad"); // string object
```

```
const r = `This is a program
          on strings in java script`;
```

```
const longString = "This is a very long string which needs \
                    to wrap across multiple lines because \
                    otherwise, my code is unreadable.";
```

string methods are given below:

String length	String slice()	String substring()	String substr()
String replace()	String replaceAll()	String toUpperCase()	String toLowerCase()
String concat()	String trim()	String trimStart()	String trimEnd()
String padStart()	String padEnd()	String charAt()	String charCodeAt()
String split()	String splice()	repeat()	

Ex-1:

```
function reverseString(str) {
  return str.split("").reverse().join("");
}
```

```
reverseString('string'); // output: gnirts
```

Ex-2:

```
<script>
  function alphabet_order(str)
  {
```

```
        return str.split("").sort().join("");
    }
    document.write(alphabet_order("datascience"));
```

</script>

Ex-3://slice(start, end)

```
var text="excellent"
text.slice(0,4) //returns "exce"
text.slice(2,4) //returns "ce"
```

ex-4://substring(from, to)

```
var text="excellent"
text.substring(0,4) //returns "exce"
text.substring(2,4) //returns "ce"
```

ex-5://includes()

```
var mystring = "Hello, welcome to CIC/AID";
var n = mystring.includes("CIC");
//output: True
```

Ex-6://repeat()

```
var string = "Welcome to VVIT";
string.repeat(2);
//output: Welcome to VVIT Welcome to VVIT
```

Ex-7: // charAt()

```
<script>
var str="javascript";
document.write(str.charAt(2));
</script>
```

Java script ES6 Features:

ES6 or the ECMAScript 2015 is the 6th and major edition of the ECMAScript language specification standard.

It defines the standard for the implementation of JavaScript and it has become much more popular than the previous edition ES5.

ES6 comes with significant changes to the JavaScript language.

Some of the best and most popular ES6 features that we can use in your everyday JavaScript coding.

- 1.let and const Keywords
- 2.Arrow Functions
- 3.Multi-line Strings
- 4.Default Parameters
- 5.Template Literals
- 6.Destructuring Assignment
- 7.Enhanced Object Literals
- 8.Promises
- 9.Classes
- 10.Modules

1. let and const keywords :

The keyword "let" enables the users to define variables and on the other hand, "const" enables the users to define constants.

Variables were previously declared using "var" which had function scope and were hoisted to the top.

It means that a variable can be used before declaration.

But, the "let" variables and constants have block scope which is surrounded by curly-braces "{}" and cannot be used before declaration.

2. Arrow Functions

ES6 provides a feature known as Arrow Functions. It provides a more concise syntax for writing function expressions by removing the "function" and "return" keywords.

Arrow functions are defined using the fat arrow (\Rightarrow) notation.

```
// Arrow function
```

```
let sumOfTwoNumbers = (a, b) => a + b;
```

```
console.log(sum(10, 20));           // Output 30
```

It is evident that there is no "return" or "function" keyword in the arrow function declaration.

3. Multi-line Strings

ES6 also provides Multi-line Strings. Users can create multi-line strings by using back-ticks(`).

It can be done as shown below :

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

4. Default Parameters

In ES6, users can provide the default values right in the signature of the functions. But, in ES5, OR operator had to be used.

//ES6

```
let calculateArea = function(height = 100, width = 50) {  
  
    // logic  
  
}
```

//ES5

```
var calculateArea = function(height, width) {  
    height = height || 50; width = width || 80;  
    // logic  
}
```

5. Template literals are literals delimited with backtick (``) characters, allowing for [multi-line strings](#), for [string interpolation](#) with embedded expressions, and for special constructs called [tagged templates](#).

Template literals are sometimes informally called *template strings*

Template literals are enclosed by backtick (``) characters instead of double or single quotes.

```
`string text`
```

```
`string text line 1 string text line 2`
```

```
`string text ${expression} string text`
```

6. Destructuring Assignment

Destructuring is one of the most popular features of ES6. The destructuring assignment is an expression that makes it easy to extract values from arrays, or properties from objects, into distinct variables.

There are two types of destructuring assignment expressions, namely, Array Destructuring and Object Destructuring. It can be used in the following manner :

//Array Destructuring

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

//Object Destructuring

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

7. Enhanced Object Literals

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

```
function getMobile(manufacturer, model, year) {  
    return { manufacturer, model, year }  
}
```

```
getMobile("Samsung", "A03", "2022");
```


8. Promises

In ES6, Promises are used for asynchronous execution. We can use promise with the arrow function as demonstrated below.

```
var asyncCall = new Promise((resolve, reject) => {  
  
    // do something resolve(); })  
    .then(() => { console.log('DON!');  
    })
```

9. Classes

Previously, classes never existed in JavaScript. Classes are introduced in ES6 which looks similar to classes in other object-oriented languages, such as C++, Java, PHP, etc. But, they do not work exactly the same way.

ES6 classes make it simpler to create objects, implement inheritance by using the "extends" keyword and also reuse the code efficiently.

In ES6, we can declare a class using the new "class" keyword followed by the name of the class.

Ex:

```
class UserProfile {  
    constructor(firstName, lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
    getName() {  
        console.log(`The Full-Name is ${this.firstName} ${this.lastName}`);  
    }  
}
```

```
let obj = new UserProfile('John', 'Abraham');  
obj.getName();
```

// output: The Full-Name is John Smith

10. Modules

Previously, there was no native support for modules in JavaScript.

ES6 introduced a new feature called modules, in which each module is represented by a separate ".js" file.

We can use the "import" or "export" statement in a module to import or export variables, functions, classes or any other component from/to different files and modules.

```
export var num = 50;  
export function getName(fullName) {  
    //data  
};
```

```
import {num, getName} from 'module';  
console.log(num); // 50
```

JavaScript types

The set of types in the JavaScript language consists of primitive values and objects.

- **Primitive values** (immutable datum represented directly at the lowest level of the language)
 - Boolean type
 - Null type
 - Undefined type
 - Number type
 - BigInt type (it is created by add 'n' at the end)
 - String type
 - Symbol type
- **Objects** (collections of properties)

Data Types

```
graph TD; A[Data Types] --> B[Primitive]; A --> C[Non-Primitive]; B --> D[Number]; B --> E[String]; B --> F[Boolean]; B --> G[NULL]; B --> H[Undefined]; B --> I[Symbol]; C --> J[Object];
```

Primitive

Non-Primitive

Number

String

Boolean

NULL

Undefined

Symbol

Object

Scope of variable:

Scope essentially means where these variables are available for use. There are three different types of scopes.

- Local Scope
- Global Scope
- Block Scope

A variable can be declared using any one of the keywords – var, let and const.

‘var’ variables: var declarations are globally scoped or function/locally scoped.

The scope is global when a var variable is declared outside a function. This means that any variable that is declared with var outside a function block is available for use in the whole window.

var is function scoped when it is declared within a function. This means that it is available and can be accessed only within that function.

```
var greeter = "hey hi";
```

```
function newFunction() {  
    var hello = "hello";  
}
```

Here, greeter is globally scoped because it exists outside a function while hello is function scoped. So, we cannot access the variable hello outside of a function. So if we do this:

```
var tester = "hey hi";
```

```
function newFunction() {  
    var hello = "hello";  
}
```

```
console.log(hello); // error: hello is not defined
```


Hoisting: Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

This means that if we do this:

```
console.log (wish);  
var wish = "say HI"
```

it is interpreted as this:

```
var wish;  
console.log(wish); // wish is undefined  
wish = "say HI"
```

So var variables are hoisted to the top of their scope and initialized with a value of undefined.

‘let’ variables: let is now preferred for variable declaration. It's no surprise as it comes as an improvement to var declarations. It also solves the problem with var that we discussed above. Let's consider why this is so.

‘let’ is in fact block scoped. A block is a chunk of code bounded by {}. A block lives in curly braces. Anything within curly braces is a block. So, a variable declared in a block with let is only available for use within that block. Here is an example:

```
let greeting = "say Hi";
```

```
let times = 4;
```

```
if (times > 3) {
```

```
    let hello = "say Hello instead";
```

```
    console.log(hello); // "say Hello instead"
```

```
}
```

```
console.log(hello) // hello is not defined
```

Just like 'var', a variable declared with 'let' can be updated within its scope. Unlike var, a let variable cannot be re-declared within its scope. So, while this will work:

```
let greeting = "say Hi";  
greeting = "say Hello"; // error
```

However, if the same variable is defined in different scopes, there will be no error:

```
let greeting = "say Hi";  
if (true) {  
  let greeting = "say Hello";  
  console.log(greeting); // "say Hello"  
}  
console.log(greeting); // "say Hi"
```

Hoisting of 'let': Just like var, let declarations are hoisted to the top. Unlike var which is initialized as undefined, the let keyword is not initialized. So, if you try to use a let variable before declaration, you'll get a Reference Error.

'Const': Variables declared with the const maintain constant values. const declarations share some similarities with let declarations. const declarations are block scoped. Like let declarations, 'const' declarations can only be accessed within the block they were declared. 'const' cannot be updated or re-declared.

This means that the value of a variable declared with const remains the same within its scope. It cannot be updated or re-declared. So, if we declare a variable with const, we can neither do this:

```
const greeting = "say Hi";  
greeting = "say Hello instead";// error: Assignment to constant variable.
```

```
const greeting = "say Hi";  
const greeting = "say Hello";// error: Identifier greeting.....
```

Every const declaration, therefore, must be initialized at the time of declaration.

This behaviour is somehow different when it comes to objects declared with `const`. While a `const` object cannot be updated, the properties of this objects can be updated. Therefore, if we declare a `const` object as this:

```
const greeting = {  
  message: "say Hi",  
  times: 4  
}
```

while we cannot do this:

```
greeting = {  
  words: "Hello",  
  number: "five"  
} // error: Assignment to constant variable.
```

we can do this: `greeting.message = "say Hello";`

So, we conclude that:

- var declarations are globally scoped or function scoped while let and const are block scoped.
- var variables can be updated and re-declared within its scope; let variables can be updated but not re-declared; const variables can neither be updated nor re-declared.
- They are all hoisted to the top of their scope. But while var variables are initialized with undefined, let and const variables are not initialized.
- While var and let can be declared without being initialized, const must be initialized during declaration.

Objects:

A JavaScript object is a mapping between *keys* and *values*. Keys are strings (or Symbols), and *values* can be anything.

This makes objects a natural fit for [hashmaps](#).

Functions are regular objects with the additional capability of being *callable*.

An object can be – Array, Date, JSON

Ex:

```
var object1= {name:"Varun", age:20};  
var object1= {name:"Vishnu", age:23};
```

```
console.log(object1);  
console.log(object2);
```

```
var obj1 = { a: 5, b: 6 };
```

```
obj1[a] =7;
```

```
console.log(obj1) // will return the value {a: 7, b: 6}
```

Individual Object:

```
let person = {  
    firstName: 'Ravi',  
    lastName: 'xyz',  
    age: 20,  
    gender: 'male'  
};
```

Array of objects can be represented as follows:

```
let products = [  
    {name: 'iPhone', price: 900},  
    {name: 'Samsung Galaxy', price: 850},  
    {name: 'Sony Xperia', price: 700}  
];
```

Nested Object:

```
let person = {  
    firstName: 'Ravi',  
    lastName: 'xyz',  
    address: {  
        place: '2/12 Brodipet',  
        city: 'Guntur',  
        state: 'Andhra Pradesh',  
        country: 'India' }  
};
```


Shallow Copy: When a reference variable is copied into a new reference variable using the assignment operator, a shallow copy of the referenced object is created.

In simple words, a reference variable mainly stores the address of the object it refers to.

When a new reference variable is assigned the value of the old reference variable, the address stored in the old reference variable is copied into the new one.

This means both the old and new reference variable point to the same object in memory

```
var object1= {name:"Varun", age:20};  
let object2 = object1;
```

```
console.log("Original=",object1);  
console.log("Copy=", object2);
```

```
object2.name = "Vijay";
```

```
console.log("Original=",object1);  
console.log("Copy=", object2);
```

Output:

- "Original=", { age: 20, name: "Varun" }
- "Copy=", { age: 20, name: "Varun" }
- "Original=", { age: 20, name: "Vijay" }
- "Copy=", { age: 20, name: "Vijay" }

Deep Copy: Unlike the shallow copy, deep copy makes a copy of all the members of the old object, allocates separate memory location for the new object and then assigns the copied members to the new object.

In this way, both the objects are independent of each other and in case of any modification to either one the other is not affected.

Also, if one of the objects is deleted the other still remains in the memory.

Now to create a deep copy of an object in JavaScript we use **JSON.parse()** and **JSON.stringify()** methods.

```
var object1= {name:"Varun", age:20};  
let object2 = JSON.parse(JSON.stringify(object1));
```

```
console.log("Original=",object1);  
console.log("Copy=", object2);
```

```
object2.name = "Vijay";
```

```
console.log("Original=",object1);  
console.log("Copy=", object2);
```

Output:

```
Before, "Original=", { age: 20, name: "Varun"}  
      "Copy=", { age: 20, name: "Varun"}
```

```
After:  "Original=", { age: 20, name: "Varun" }  
      "Copy=", { age: 20, name: "Vijay"}
```

Object Methods:

```
var student = {  
    name: "Vijay",  
    class : "B.Tech 3rd Year",  
    section : "B",  
  
    studentDetails : function() {  
        return this.name + " " + this.class + " " + this.section + " ";  
    }  
};  
console.log(student.studentDetails())
```

String - The **String** object is used to represent and manipulate a sequence of characters.

Strings are useful for holding data that can be represented in text form.

Some of the most-used operations on strings are to check their [length](#), to build and concatenate them using the [+ and += string operators](#).

Creating strings: Strings can be created as primitives, from string literals, or as objects, using the [String\(\)](#) constructor:

```
const string1 = "A string primitive"; // using double quotes
const string2 = 'Also a string primitive'; // using single quotes
const string3 = `Yet another string primitive`; // using back ticks
```

Note: String primitives and string objects can be used interchangeably in most situations.

There are two ways to access an individual character in a string. The first is the `charAt()` method:

```
return 'cat'.charAt(1) // returns "a"
```

The other way (introduced in ECMAScript 5) is to treat the string as an array-like object, where individual characters correspond to a numerical index:

```
return 'cat'[1] // returns "a"
```

```
//strings example
```

```
const name1 = 'Dhoni';
```

```
const name2 = "Raina";
```

```
const result = `The names are ${name1} and ${name2} `;
```

List of String Methods:

<code>bold()</code>	<code>charAt()</code>	<code>concat()</code>	<code>endsWith()</code>	<code>fontcolor()</code>
<code>includes()</code>	<code>indexOf()</code>	<code>lastIndexOf()</code>	<code>match()</code>	<code>matchAll()</code>
<code>repeat()</code>	<code>replace()</code>	<code>replaceAll()</code>	<code>search()</code>	<code>slice()</code>
<code>split()</code>	<code>startsWith()</code>	<code>substr()</code>	<code>substring()</code>	<code>toLowerCase()</code>
<code>toString()</code>	<code>toUpperCase()</code>	<code>trim()</code>		

Ex:

```
const text1 = 'hello'; const text2 = 'world';  
const text3 = ' JavaScript '; // concatenating two strings  
const result1 = text1.concat(' ', text2);  
console.log(result1); // "hello world"
```

```
// converting the text to uppercase  
const result2 = text1.toUpperCase();  
console.log(result2); // HELLO
```

```
// removing whitespace from the string  
const result3 = text3.trim();  
console.log(result3); // JavaScript
```

```
// converting the string to an array  
const result4 = text1.split();  
console.log(result4);
```

```
// ["hello"] // slicing the string  
const result5 = text1.slice(1, 3);  
console.log(result5); // "el"
```

substring():

1) Extracting a substring from the beginning of the string example

```
let str = 'JavaScript Substring';  
let substring = str.substring(0,10);  
console.log(substring);
```

Output: JavaScript

2) Extracting a substring to the end of the string example

```
let str = 'JavaScript Substring';  
let substring = str.substring(11);  
console.log(substring);
```

Output: Substring

3) Extracting domain from the email example. The following example uses the substring() with the indexOf() to extract the domain from the email:

```
let email = 'varun.k@gmail.com';  
let domain = email.substring(email.indexOf('@') + 1);  
  
console.log(domain); // gmail.com
```

repeat():

```
let x = 'Krishna';  
console.log(x.repeat(3));
```

Output: Krishna Krishna Krishna

Escape Character in Strings: You can use the backslash escape character \ to include special characters in a string.

For example,

```
const name = 'My name is \'Peter\'.';  
console.log(name);
```

Output: My name is 'Peter'.

Functions in Java script:

Functions are one of the fundamental building blocks in JavaScript.

A function in JavaScript is similar to a procedure—a set of statements that performs a task or calculates a value, but for a procedure to qualify as a function, it should take some input and return an output where there is some obvious relationship between the input and the output.

To use a function, you must define it somewhere in the scope from which you wish to call it.

Function declarations

A **function definition** (also called a **function declaration**, or **function statement**) consists of the function keyword, followed by:

- The name of the function.
- A list of parameters to the function, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, enclosed in curly brackets, {...}.

Types of functions:

- 1) Normal functions
- 2) Arrow Functions
- 3) Anonymous Functions
- 4) Nested functions
 - Closures
- 5) Recursive Functions

Syntax:

```
function functionName(parameters) {  
  
    // function body  
    // ...  
  
}
```

Ex:

```
function myfun(message) {  
    console.log(message);  
}
```

```
let result = myfun('Hello');  
console.log('Result:', result);
```

```
// 'Hello' is argument  
// message is parameter
```

```
function display(message) {  
  
    if (! message ) {  
        return;  
    }  
    console.log(message);  
}
```

Arguments Object:

```
function add() {  
    let sum = 0;  
    for (let i = 0; i < arguments.length; i++) {  
        sum += arguments[i];  
    }  
    return sum;  
}
```

```
console.log(add(1, 2));           // 3  
console.log(add(1, 2, 3, 4, 5)); // 15  
console.log(add(4,7,2,1,3,6));   // 23
```

The arguments variable is an array-like object inside a function, representing function arguments

Storing functions in variables

Functions are first-class citizens in JavaScript. In other words, you can treat functions like values of other types. The following defines the `add()` function and assigns the function name to the variable `sum`:

```
function add(a, b) {  
  return a + b;  
}
```

```
let sum = add;
```

```
console.log(sum(5,6));  
console.log(add(7,8));
```

Passing a function to another function

Because functions are values, you can pass a function as an argument into another function.

```
function add(a, b) {  
    return a + b;  
}
```

```
let sum = add;
```

```
function average(a, b, myfun) {  
    return myfun(a, b) / 2;  
}
```

```
let result = average(10, 20, sum);
```

```
console.log(result);
```

```
var firstWord = "Mary";  
var secondWord = "Army";
```

```
console.log( isAnagram(firstWord, secondWord)) ; // true
```

```
function isAnagram(first, second) {
```

```
    var a = first.toLowerCase();  
    var b = second.toLowerCase();
```

```
    a = a.split("").sort().join("");  
    b = b.split("").sort().join("");
```

```
    return a === b;  
}
```


Rest operator or Rest parameter:

ES6 introduced two new operators, [spread](#) and rest operators. The rest operator is used to create a function that accepts a variable number of arguments in JavaScript. It is a collection of all remaining elements

Note: Rest operator is also known as Rest parameter.

```
function functionName(...parameterName){  
    //function body  
}
```

Here, parameterName is a rest parameter. It is preceded by three dots (...).

There are essential points that you must remember regarding rest parameters:

1. Apart from other parameters, a function can have only one rest parameter.
2. The rest parameter is an array. You can use loops such as `forEach()`, `for/of`, and others to iterate through it.

Ex:

```
function display(arg1, arg2, ...args){  
    console.log(arg1);  
    console.log(arg2);  
    console.log(args);  
}  
display(2, 3, 4, 5, 6);
```

Output:

2

3

4 5 6

```
var myName = ["VidyaSagar" , "Vasireddy" , "VVIT"] ;  
const [firstName , ...familyName] = myName ;  
console.log(firstName); // VidyaSagar ;  
console.log(familyName); // [ "Vasireddy" , "VVIT" ] ;
```

Spread Operator [...spread]:

It's the opposite to rest parameter , where rest parameter collects items into an array, the spread operator unpacks the collected elements into single elements.

Ex-1

```
var myName = ["VidyaSagar" , "Vasireddy" , "VVIT"];  
var newArr = [...myName , "VIVA" , 2007];  
console.log(newArr) ; // ["VidyaSagar" , "Vasireddy" , "VVIT" , "VIVA" , 2007];
```

Ex-2

```
function sum(x, y, z) {  
    return x + y + z;  
}  
  
const numbers = [1, 2, 3];  
console.log(sum(...numbers)); // expected output: 6
```

Anonymous Functions: It is a name less function. An anonymous function is a function without a name.

The following shows how to define an anonymous function:

```
(function () {  
    //...  
});
```

```
function (a) {  
    return a + 100;  
}
```

```
function (a, b){  
    return a + b + 100;  
}
```

An anonymous function is not accessible after its initial creation. Therefore, you often need to assign it to a variable.

For example, the following shows an anonymous function that displays a message:

```
let show = function() {  
    console.log('it is my Anonymous function');  
};
```

```
show();
```

Nested functions and closures:

You may nest a function within another function. The nested (inner) function is private to its containing (outer) function. It also forms a *closure*.

A closure is an expression (most commonly, a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).

Since a nested function is a closure, this means that a nested function can "inherit" the arguments and variables of its containing function.

In other words, the inner function contains the scope of the outer function. To summarize:

- The inner function can be accessed only from statements in the outer function.
- The inner function forms a closure: the inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function.

The following example shows nested functions:

Ex-1:

```
function addSquares(a, b) {  
  function square(x) {  
    return x * x;  
  }  
  return square(a) + square(b);  
}  
a = addSquares(2, 3); // returns 13  
b = addSquares(3, 4); // returns 25  
c = addSquares(4, 5); // returns 41
```

Ex-2:

```
function outside(x) {  
  function inside(y) {  
    return x + y;  
  }  
  return inside;  
}  
  
fn_inside = outside(3);  
  
result = fn_inside(5); // returns 8  
  
result1 = outside(3)(5); // returns 8
```

Ex-3:

```
function A(x) {  
    function B(y) {  
        function C(z) {  
            console.log(x + y + z);  
        }  
        C(3);  
    }  
    B(2);  
}
```

```
Console.log(A(1));           // logs 6 (1 + 2 + 3)
```


Closures

Closures are one of the most powerful features of JavaScript.

- JavaScript variables can belong to the **local** or **global** scope.
- Global variables can be made local (private) with **closures**.

JavaScript allows for the nesting of functions and grants the inner function full access to all the variables and functions defined inside the outer function (and all other variables and functions that the outer function has access to).

However, the outer function does *not* have access to the variables and functions defined inside the inner function.

This provides a sort of encapsulation for the variables of the inner function.

Ex-1:

```
function greeting() {  
  let message = 'Hi';  
  
  function sayHi() {  
    console.log(message);  
  }  
  
  sayHi();  
}  
  
greeting();
```

Ex-2:

```
const add = ( function () {  
  let counter = 0;  
  return function () {  
    counter += 1;  
    return counter;  
  }  
} ) ();  
  
add();  
add();  
add();
```

Ex-3:

```
function greeting() {  
  
    let message = 'Hi';  
    function sayHi() {  
        console.log(message);  
    }  
    return sayHi; // returning a function  
}  
  
let hi = greeting();  
  
hi(); // still can access the message variable
```

Ex-4:

```
function greeting(message) {  
  
    return function(name) {  
        return message + ' ' + name;  
    }  
}  
  
let sayHi = greeting('Hi');  
let sayHello = greeting('Hello');  
  
console.log(sayHi('John')); // Hi John  
console.log(sayHello('John')); // Hello John
```

In fact, closures provide a means of creating class-like privacy similar to that used in Object Oriented Programming, allowing us to emulate private methods.

```
var makeCounter = function() {  
  let count = 0;  
  function changeBy(val) {  
    count += val;  
  }  
  return {  
    increment: function() {  
      changeBy(1);  
    },  
    decrement: function() {  
      changeBy(-1);  
    },  
    value: function() {  
      return count;  
    }  
  }  
};
```

```
var counter1 = makeCounter();  
var counter2 = makeCounter();  
  
console.log(counter1.value()); // returns 0  
  
counter1.increment(); // adds 1  
counter1.increment(); // adds 1  
  
console.log(counter1.value()); // returns 2  
  
counter1.decrement(); //subtracts 1  
  
console.log(counter1.value()); // returns 1  
console.log(counter2.value()); // returns 0
```

Currying:

Currying is the pattern of functions that immediately evaluate and return other functions. This is made possible by the fact that JavaScript functions are expressions that can return other functions.

Curried functions are constructed by chaining closures by defining and immediately returning their inner functions simultaneously.

In other terms, currying is when a function — instead of taking all arguments at one time — takes the first one and returns a new function, which takes the second one and returns a new function, which takes the third one, etc. until all arguments are completed.

Currying is a function that takes one argument at a time and returns a new function expecting the next argument.

It is a transformation of functions that translates a function from callable as $f(a, b, c)$ into callable as $f(a)(b)(c)$.

Definition: Currying simply means evaluating functions with multiple arguments and decomposing them into a sequence of functions with a single argument.

There are several reasons why currying is ideal:

1. Currying is a checking method to make sure that you get everything you need before you proceed
2. It helps you to avoid passing the same variable again and again
3. It divides your function into multiple smaller functions that can handle one responsibility. This makes your function pure and less prone to errors and side effects
4. It is used in functional programming to create a higher-order function
5. This could be personal preference, but I love that it makes my code readable

Non-curried version

```
const add = (a, b, c) => {  
  return a + b + c  
}  
console.log(add(2, 3, 5)) // 10
```

Curried version

```
const addCurry = (a) => {  
  return (b) => {  
    return (c) => {  
      return a + b + c  
    }  
  }  
}  
console.log(addCurry(2)(3)(5)) // 10
```

Ex:

```
let greeting = function (a) {  
  return function (b) {  
    return a + ' ' + b  
  }  
}
```

```
let hello = greeting('Hello')  
let morning = greeting('Good morning')
```

```
hello('Austin') // returns Hello Austin  
hello('Roy') // returns Hello Roy  
morning('Austin') // returns Good morning  
Austin  
morning('Roy') //returns Good Morning Roy
```

Arrow Functions:

An **arrow function expression** is a compact alternative to a traditional **function expression**, but is limited and can't be used in all situations.

```
let add = function (x, y) {  
    return x + y;  
};  
  
console.log(add(10, 20)); // 30
```

```
let add = (x, y) => x + y;  
console.log(add(10, 20)); // 30;
```

```
let add = (x, y) => { return x + y; };
```

```
hello = ( ) => {  
    return "Hello World!";  
}
```

```
var hello;  
  
hello = val => "Hello " + val;  
  
Console.log(hello("Universe!"));
```


There are differences between *arrow functions* and *traditional functions*, as well as some limitations:

1. Arrow functions don't have their own bindings to [this](#) or [super](#), and should not be used as [methods](#).
2. Arrow functions don't have access to the [new.target](#) keyword.
3. Arrow functions aren't suitable for [call](#), [apply](#) and [bind](#) methods, which generally rely on establishing a [scope](#).
4. Arrow functions cannot be used as [constructors](#).
5. Arrow functions cannot use [yield](#), within its body.

Array

The **Array** object, as with arrays in other programming languages, enables storing a collection of multiple items under a single variable name, and has members for performing common array operations.

- JavaScript arrays are **resizable** and can contain a mix of different [data types](#).
- JavaScript arrays are **not associative arrays** and so, array elements cannot be accessed using strings as indexes, but must be accessed using integers as indexes.
- JavaScript arrays are [zero-indexed](#).
- JavaScript [array-copy operations](#) create [shallow copies](#).

Array Methods:

- static methods
- iterative methods

The following are the static methods:

[Array.from\(\)](#): Creates a new Array instance from an array-like object or iterable object.

[Array.isArray\(\)](#): Returns true if the argument is an array, or false otherwise.

[Array.of\(\)](#): Creates a new Array instance with a variable number of arguments, regardless of number or type of the arguments.

```
Array.of(7);           // [7]
Array(7);              // array of 7 empty slots
Array.of(1, 2, 3);     // [1, 2, 3]
Array(1, 2, 3);        // [1, 2, 3]
Array.of(1);           // [1]
Array.of(1, 2, 3);     // [1, 2, 3]
Array.of(undefined);   // [undefined]
```

Instance Methods:

find()	at()	Fill()	forEach()	from()	includes()
join()	map()	pop()	push()	reduce()	reverse()
slice()	sort()	toString()	some()	filter()	concat()

Concat(): The **concat()** method is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array.

Ex:

```
const array1 = ['a', 'b', 'c'];  
const array2 = ['d', 'e', 'f'];  
const array3 = array1.concat(array2);  
console.log(array3); // expected output: Array ["a", "b", "c", "d", "e", "f"]
```

fill():

The **fill()** method changes all elements in an array to a static value, from a start index (default 0) to an end index (default array.length). It returns the modified array.

ex:

```
const array1 = [1, 2, 3, 4]; // fill with 0 from position 2 until position 4  
console.log(array1.fill(0, 2, 4)); // expected output: [1, 2, 0, 0]
```

Java script Arrays

The **Array** object, as with arrays in other programming languages, enables storing a collection of multiple items under a single variable name, and has members for performing common array operations.

- JavaScript arrays are resizable **and** can contain a mix of different **data types**.
- JavaScript arrays are **zero-indexed**
- JavaScript **array-copy operations** create **shallow copies**.

Array Methods:

The following are the static methods:

[Array.from\(\)](#): Creates a new Array instance from an array-like object or iterable object.

[Array.isArray\(\)](#): Returns true if the argument is an array, or false otherwise.

[Array.of\(\)](#): Creates a new Array instance with a variable number of arguments, regardless of number or type of the arguments.

Array.of() method creates a new Array instance from a variable number of arguments, regardless of number or type of the arguments.

The difference between **Array.of()** and the **Array** constructor is in the handling of integer arguments: **Array.of(7)** creates an array with a single element, 7, whereas **Array(7)** creates an empty array with a length property of 7

```
Array.of(7); // [7]
Array(7); // array of 7 empty slots
```

```
Array.of(1, 2, 3); // [1, 2, 3]
Array(1, 2, 3); // [1, 2, 3]
```

```
Array.of(1); // [1]
Array.of(1, 2, 3); // [1, 2, 3]
Array.of(undefined); // [undefined]
```

Instance Methods:

Find()	at()	Fill()	forEach()	from()	includes()
join()	map()	pop()	push()	reduce()	reverse()
slice()	sort()	toString()	some()	filter():	

Ex-1:

```
function isBigEnough(value) {
  return value >= 10
}

let filtered = [12, 5, 8, 130, 44].filter(isBigEnough)
// filtered is [12, 130, 44]
```

Ex-2: The following example returns all prime numbers in the array:

```
const array = [-3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13];
```

```
function isPrime(num) {  
  for (let i = 2; num > i; i++) {  
    if (num % i == 0) {  
      return false;  
    }  
  }  
  return num > 1;  
}
```

```
console.log(array.filter(isPrime)); // [2, 3, 5, 7, 11, 13]
```

Ex-3:

```
let fruits = ['apple', 'banana', 'grapes', 'mango', 'orange']
```

```
/**  
 * Filter array items based on search criteria (query)  
 */  
function filterItems(arr, query) {  
  return arr.filter(function(el) {  
    return el.toLowerCase().indexOf(query.toLowerCase()) !== -1  
  })  
}
```

```
console.log(filterItems(fruits, 'ap')) // ['apple', 'grapes']  
console.log(filterItems(fruits, 'an')) // ['banana', 'mango', 'orange']
```

join(): The **join()** method creates and returns a new string by concatenating all of the elements in an array (or an [array-like object](#)), separated by commas or a specified separator string. If the array has only one item, then that item will be returned without using the separator.

Ex:

```
const elements = ['Fire', 'Air', 'Water'];  
  
console.log(elements.join());    // expected output: "Fire,Air,Water"  
console.log(elements.join("")); // expected output: "FireAirWater"  
console.log(elements.join("-")); // expected output: "Fire-Air-Water"
```

at(): returns value at the given index location.

Ex:

```
const array1 = [5, 12, 8, 130, 44];  
  
let index = 2;  
  
console.log(`Using an index of ${index} the item returned is ${array1.at(index)}`);  
  
// expected output: "Using an index of 2 the item returned is 8"
```

```
index = -2;
```

```
console.log(`Using an index of ${index} item returned is ${array1.at(index)}`);
```

```
// expected output: "Using an index of -2 item returned is 130"
```

Concat(): The **concat()** method is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array.

Ex:

```
const array1 = ['a', 'b', 'c'];
```

```
const array2 = ['d', 'e', 'f'];
```

```
const array3 = array1.concat(array2);
```

```
console.log(array3); // expected output: Array ["a", "b", "c", "d", "e", "f"]
```

fill(): The **fill()** method changes all elements in an array to a static value, from a start index (default 0) to an end index (default array.length). It returns the modified array.

ex:

```
const array1 = [1, 2, 3, 4]; // fill with 0 from position 2 until position 4
```

```
console.log(array1.fill(0, 2, 4)); // expected output: [1, 2, 0, 0]
```

```
// fill with 5 from position 1
```

```
console.log(array1.fill(5, 1)); // expected output: [1, 5, 5, 5]
```

```
console.log(array1.fill(6)); // expected output: [6, 6, 6, 6]
```

find(): The **find()** method returns the first element in the provided array that satisfies the provided testing function. If no values satisfy the testing function, [undefined](#) is returned.

Ex:

```
const array1 = [5, 12, 8, 130, 44];
```

```
const found = array1.find(element => element > 10);
```

```
console.log(found); // expected output: 12
```

forEach(): The **forEach()** method executes a provided function once for each array element.

Syntax:

```
// Arrow function
```

```
forEach((element) => { /* ... */ })
```

```
forEach((element, index) => { /* ... */ })
```

```
forEach((element, index, array) => { /* ... */ })
```

Ex:

```
const array1 = ['a', 'b', 'c'];
```

```
array1.forEach(element => console.log(element));
```


Higher Order Functions

In JavaScript, functions are a data type just as strings, numbers, and arrays are data types. Therefore, functions can be assigned as values to variables, but are different from all other data types because they can be invoked.

In Java script, functions can be assigned to variables in the same way that strings or arrays can. They can be passed into other functions as parameters or returned from them as well.

A “higher-order function” is **a function that accepts functions as parameters and/or returns a function.**

JavaScript functions are first-class objects.

Therefore:

- They have built-in properties and methods, such as the name property and the `.toString()` method.
- Properties and methods can be added to them.
- They can be passed as arguments and returned from other functions.
- They can be assigned to variables, array elements, and other objects.

forEach():

The forEach() method calls a function for each element in an array.
The forEach() method is not executed for empty elements.

The forEach method passes a [callback function](#) for each element of an array together with the following parameters:

- Current Value (required) - The value of the current array element
- Index (optional) - The current element's index number
- Array (optional) - The array object to which the current element belongs

<i>function()</i>	Required. A function to run for each array element.
<i>currentValue</i>	Required. The value of the current element.
<i>index</i>	Optional. The index of the current element.
<i>arr</i>	Optional. The array of the current element.
<i>thisValue</i>	Optional. Default undefined. A value passed to the function as its this value.

Return value: undefined

Ex-1:

```
let sum = 0;
const numbers = [65, 44, 12, 4];
numbers.forEach(myFunction);

function myFunction(item) {
  sum += item;
}
Console.log(sum);
```

Ex-2:

```
let students = ['Vijay', 'Varun', 'Viswa'];

students.forEach(myFunction);

function myFunction(item) {
  console.log(item);
}
```

```
// Task-Durations are in minutes
```

```
const tasks = [  
  { 'name'    : 'Lunch', 'duration' : 40 },  
  { 'name'    : 'Work out', 'duration' : 60 },  
  { 'name'    : 'Job', 'duration' : 480 },  
  { 'name'    : 'Sleep', 'duration' : 360 }  
];
```

```
const task_names = [];  
tasks.forEach(function (task) {  
  task_names.push(task.name);  
});
```

```
console.log(task_names)
```


filter()

1. The filter() method creates a new array filled with elements that pass a test provided by a function.
2. The filter() method does not execute the function for empty elements.
3. The filter() method does not change the original array.

Syntax:

array.filter(function(currentValue, index, arr), thisValue)

<i>function()</i>	Required. A function to run for each array element.
<i>currentValue</i>	Required. The value of the current element.
<i>index</i>	Optional. The index of the current element.
<i>arr</i>	Optional. The array of the current element.
<i>thisValue</i>	Optional. Default undefined A value passed to the function as its this value.

return value:

An array	Containing the elements that pass the test. If no elements pass the test it returns an empty array.
----------	--

/* Example for filter(): to list out words with less than 8 letters */

Ex-1:

```
function filterDemo() {  
  
    const words = ['Python', 'Javascript', 'Go', 'Java', 'PHP', 'Ruby'];  
  
    const result = words.filter(word => word.length < 8);  
  
    console.log(result);  
}
```

Example-2:

```
const tasks = [  
  { 'name' : 'Lunch', 'duration' : 40 },  
  { 'name' : 'Work out', 'duration' : 60 },  
  { 'name' : 'Job', 'duration' : 480 },  
  { 'name' : 'Sleep', 'duration' : 360 }  
];
```

```
const difficult_tasks = tasks.filter((task) => task.duration >= 120 );
```

map()

Java script **map()** creates a new array, which contains the results obtained from iterating over the elements of the specified array and calling the provided function *once* for each element in order.

Note: **map()** only works on arrays. It does not execute a function for an empty array.

Syntax:

```
const newArray = oldArray.map((currentValue, index, array)=> {  
    // Do stuff with currentValue  
}  
);
```

Ex-1: //creating an array

```
var my_array = [1, 3, 5, 2, 4];
```

//map calls a function which has "item" passed as parameter. map will pass each element of my_array as "item" in this function. The function will double each element passed to it and return it

```
result = my_array.map(function(item) {  
    return item*2;  
});
```

```
//prints new list containing the doubled values  
console.log(result);
```

Ex-2:

```
const numbers = [9, 36, 64, 144];
```

```
let squareRoots = numbers.map((number) => {  
    return Math.sqrt(number);  
});
```

```
console.log(squareRoots); // [3,6,8,12]
```

Ex-3:

```
const names = ["Pune", "Hyderabad", "Kochin", "Delhi"];
```

```
let lengths = names.map((name) => name.length);
```

```
console.log(lengths); // [4, 9, 6, 5]
```

Ex-4:

```
const students = [  
    {firstName : "Vijay", lastName: "Vasireddy"},  
    {firstName : "Rohith", lastName: "Vinjamuri"},  
    {firstName : "Varun", lastName: "Kompalli"}  
];  
  
let studentsNames = students.map(student => {  
    return `${student.firstName} ${student.lastName}`;  
});  
  
console.log(studentsNames);
```


Map() Vs forEach()

The main distinction between these two methods is that the `map()` function returns *a new array*, whilst the `forEach()` method does not - it *alters the original array*.

reduce():

The **reduce()** method executes a user-supplied "reducer" callback function on each element of the array, in order, passing in the return value from the calculation on the preceding element. The final result of running the reducer across all elements of the array is a single value.

The reduce() method:

- executes a reducer function for array element.
- returns a single value: the function's accumulated result.
- does not execute the function for empty array elements.
- does not change the original array.

Ex-1:

```
const a = [33,66,44,88,11];  
console.log(a.reduce((a,b)=> (a>b)?a:b));
```

Ex-2:

```
var arr = [10, 20, 30, 40, 50, 60];  
  
    function sumofArray(sum, num) {  
        return sum + num;  
    }  
    console.log(arr.reduce(sumofArray));
```

Ex-3:

```
let flattened = [[0, 1], [2, 3], [4, 5]].reduce( function(previousValue, currentValue) {  
  
    return previousValue.concat (currentValue) }, [ ] )
```

Output: flattened is [0, 1, 2, 3, 4, 5]

Ex-4:

```
const names = ['Alice', 'Bob', 'Tiff', 'Bruce', 'Alice'];
```

```
let countedNames = names.reduce(function (allNames, name) {
```

```
    if (name in allNames)
```

```
        allNames[name]++;
```

```
    else
```

```
        allNames[name] = 1;
```

```
    return allNames;
```

```
}, {})
```

```
console.log(countedNames);
```

Output:

```
{  
  Alice:2,  
  Bob:1,  
  Tiff:1,  
  Bruce:1  
}
```

OOPs in Java script

There are certain features or mechanisms which makes a Language Object-Oriented like:

- **Object**
- **Classes**
- **Encapsulation**
- **Inheritance**

1. Object– An Object is a **unique** entity that contains **property** and **methods**.

- For example “car” is a real life Object, which has some characteristics like color, type, model, horsepower and performs certain action like drive.
- The characteristics of an Object are called as Property, in Object-Oriented Programming and the actions are called methods.
- An Object is an **instance** of a class.

Ex:

```
let person = {  
  first_name:'VidyaSagar',  
  last_name: 'Vasireddy',  
  
  //method  
  getFunction : function(){  
    return (`The name of the person is ${person.first_name}  
            ${person.last_name}`);  
  },  
  //object within object  
  phone_number : {  
    mobile:'12345',  
    landline:'6789'  
  }  
}  
  
console.log(person.getFunction());  
console.log(person.phone_number.landline);
```


Classes are **blueprint** of an Object.

A class can have many Object, because class is a **template** while Object are **instances** of the class or the concrete implementation.

Before we move further into implementation, we should know unlike other Object Oriented Language there is **no classes in JavaScript** we have only Object.

To be more precise, JavaScript is a prototype based object oriented language, which means it doesn't have classes rather it define behaviors using constructor function and then reuse it using the prototype.

```
class Vehicle {  
  constructor(name, maker, engine) {  
    this.name = name;  
    this.maker = maker;  
    this.engine = engine;  
  }  
  getDetails(){  
    return (`The name of the bike is ${this.name}.`)  
  }  
}  
  
// Making object with the help of the constructor  
let bike1 = new Vehicle('Maruti', 'Baleno', '1200cc');  
let bike2 = new Vehicle('Kia', 'Seltos', '1400cc');  
  
console.log(bike1.name); // Hayabusa  
console.log(bike2.maker); // Kawasaki  
console.log(bike1.getDetails());
```

3. Encapsulation – The process of **wrapping property and function** within a **single unit** is known as encapsulation.

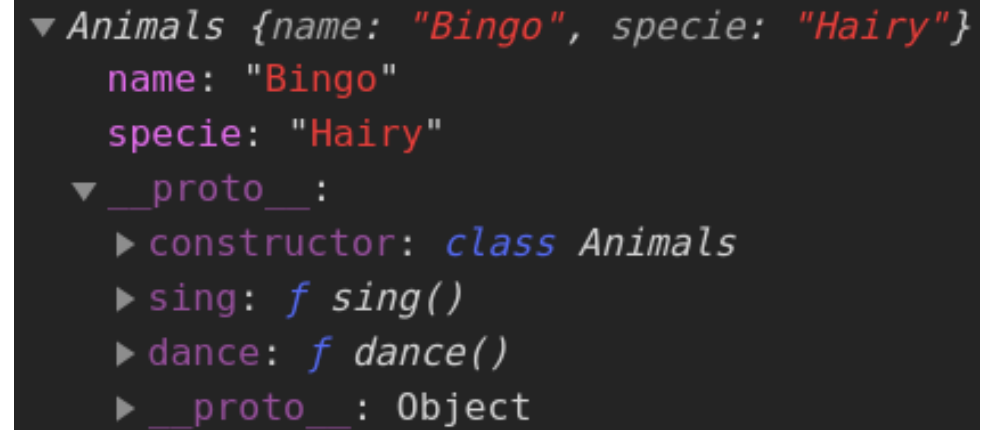
4. Inheritance – It is a concept in which some property and methods of an Object is being used by another Object.

Unlike most of the OOP languages where classes inherit classes, JavaScript Object inherits Object i.e. certain features (property and methods) of one object can be reused by other Objects.

```
class Animals {  
  constructor(name, specie) {  
    this.name = name;  
    this.specie = specie;  
  }  
  sing() {  
    return `${this.name} can sing`;  
  }  
  dance() {  
    return `${this.name} can dance`;  
  }  
}
```

```
let bingo = new Animals("Bingo", "Hairy");  
console.log(bingo);
```

This is the result in the console:



```
▼ Animals {name: "Bingo", specie: "Hairy"}  
  name: "Bingo"  
  specie: "Hairy"  
  ▼ __proto__:  
    ► constructor: class Animals  
    ► sing: f sing()  
    ► dance: f dance()  
    ► __proto__: Object
```

The `__proto__` references the `Animals` prototype (which in turn references the `Object` prototype).

From this, we can see that the constructor defines the major features while everything outside the constructor (`sing()` and `dance()`) are the bonus features (**prototypes**).

In the background, using the new keyword approach, the above translates to:

```
function Animals(name, specie) {  
    this.name = name; this.specie = specie;  
}  
Animals.prototype.sing = function(){  
    return `${this.name} can sing`;  
}  
Animals.prototype.dance = function() {  
    return `${this.name} can dance`;  
}  
let Bingo = new Animals("Bingo", "Hairy");
```

```
Ex: class Animal {  
  constructor(legs) {  
    this.legs = legs;  
  }  
  walk() {  
    console.log('walking on ' + this.legs + ' legs');  
  }  
}
```

```
class Bird extends Animal {  
  constructor(legs) {  
    super(legs);  
  }  
  fly() {  
    console.log('flying');  
  }  
}
```

```
let bird = new Bird(2);  
bird.walk();  
bird.fly();
```

Output:
"walking on 2 legs"
"flying"

Java script Event Handling

An HTML event can be something the browser does, or something a user does.

Here are some examples of HTML events:

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

Often, when events happen, you may want to do something.

JavaScript lets you execute code when events are detected.

HTML allows event handler attributes, **with JavaScript code**, to be added to HTML elements.

The change in the state of an object is known as an **Event**. In html, there are various events which represents that some activity is performed by the user or by the browser.

When [java script](#) code is included in [HTML](#), js react over these events and allow the execution. This process of reacting over the events is called **Event Handling**. Thus, js handles the HTML events via **Event Handlers**.

For example, when a user clicks over the browser, add js code, which will execute the task to be performed on the event.

Most oftenly used events are: **Mouse Events**, **Keyboard Events**, **Document/Window Event**, Form Events

Event Performed	Event Handler	Description
click	onclick	When mouse click on an element
mouseover	onmouseover	When the cursor of the mouse comes over the element
mouseout	onmouseout	When the cursor of the mouse leaves an element
mousedown	onmousedown	When the mouse button is pressed over the element
mouseup	onmouseup	When the mouse button is released over the element
mousemove	onmousemove	When the mouse movement takes place.

Event Performed	Event Handler	Description
Keydown & Keyup	onkeydown & onkeyup	When the user press and then release the key

Event Performed	Event Handler	Description
focus	onfocus	When the user focuses on an element
submit	onsubmit	When the user submits the form
blur	onblur	When the focus is away from a form element
change	onchange	When the user modifies or changes the value of a form element

Event Performed	Event Handler	Description
focus	onfocus	When the user focuses on an element
submit	onsubmit	When the user submits the form
blur	onblur	When the focus is away from a form element
change	onchange	When the user modifies or changes the value of a form element

```
<!DOCTYPE html>
```

```
<html>
```

```
<body>
```

```
<button onclick="document.getElementById('demo').innerHTML=Date()">
```

```
Date and Time
```

```
</button>
```

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

```
</body>
```

```
</html>
```



```
<!DOCTYPE HTML>
<html>
<head>
<style>
#div1 {
  width: 350px;
  height: 70px;
  padding: 10px;
  border: 1px solid #aaaaaa;
}
</style>
```

```
<script>
function allowDrop(ev) {
  ev.preventDefault();
}

function drag(ev) {
  ev.dataTransfer.setData("text", ev.target.id);
}

function drop(ev) {
  ev.preventDefault();
  var data = ev.dataTransfer.getData("text");

  ev.target.appendChild(document.getElementById(data));
}
</script>
</head>
```

```
<body>
```

```
<p>Drag the image into the rectangle:</p>
```

```
<div id="div1" ondrop="drop(event)" ondragover="allowDrop(event)"></div>
```

```
<div id="div1" ondrop="drop(event)" ondragover="allowDrop(event)"></div>
```

```
<div id="div1" ondrop="drop(event)" ondragover="allowDrop(event)"></div>
```

```
<br>
```

```
<div id="drag1" draggable="true" ondragstart="drag(event)" >Sri Harsha</div>
```

```
<div id="drag2" draggable="true" ondragstart="drag(event)" >Mohan Krishna</div>
```

```
<div id="drag3" draggable="true" ondragstart="drag(event)" >Hari </div>
```

```
</body>
```

```
</html>
```