[**Variables**](https://developer.mozilla.org/en-US/docs/Glossary/Variable) **::** are containers that store values. Variables created **without** a declaration keyword (var, let, or const) are always global, even if they are created inside a function.

**Spread ::** "expands" an array into its elements.

**Rest ::** collects multiple elements and "condenses" them into a single element.

[**Functions**](https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/JavaScript_basics#functions) **::** a way of packaging functionality that you wish to reuse.

### [Events](https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/JavaScript_basics#events) :: Events are actions that happen when a user interacts with the page - like clicking an element, typing in a field, or loading a page.

**Hoisting ::**

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

**deep copying** **::** A deep copy means that all of the values of the new variable are copied and **disconnected from the original**variable.

**shallow copying** **::** A shallow copy means that certain (sub-)values are **still connected to the original** variable.

**Synchronous JS::** three images are in the same lane. One can't overtake the other. The race is finished one by one. If image number 2 stops, the following image stops.



Synchronous code in JavaScript:

console.log(" I ");

console.log(" Eat ");

console.log(" Ice Cream ");

OUTPUT ::



**Asynchronous js ::** the three images are in different lanes. They'll finish the race on their own pace. Nobody stops for anybody

****

ASynchronous code in JavaScript:

console.log("I");

setTimeout(()=>{

 console.log("Eat"); // This will be shown after 2 seconds

},2000)

console.log("Ice Cream")

OUTPUT ::



**Mechanisms of asynchronous**

* **JavaScript using 3 techniques::**

**1. Callbacks ::**

When you nest a function inside another function as an argument, that's called a callback. OR

Callbacks are functions passed as arguments into other functions to make sure mandatory variables are available within the callback-function's scope.

**2 Promises ::**

Promises are placeholder objects for data that's available in the future.

promise has three states:

1. **Pending :: didn’t order yet**
2. **Resolved** :: order received
3. **Rejected :: customer didn’t received**

# Async / Await ::

The keyword Await makes JavaScript wait until a promise settles and returns its result.

## Different phases of events –

## Capture :: (also called the trickling phase)

## the event "trickles down" to the element that caused the event.

It starts from the root level element and handler,

and then propagates down to the element. (Shortcut : Trickle Down)

**Target ::** The capture phase is completed when the event reaches the Target.

**Bubble ::** the event is "bubbled" up to the DOM tree.

It is first captured and handled by the innermost handler (the one that is closest to the element on which the event occurred). It then bubbles up (or propagates up) to the higher levels of DOM tree, further up to its parents, and then finally to its root.(Shortcut : Bubble Up)

**Higher Order Function::**

Higher Orders Functions are functions that perform operations on other functions.

 OR

In this definition, operations can mean taking one or more functions as an argument OR returning a function as the result. It doesn't have to do both. Doing one or the other qualifies a function as a higher order function.

**two types of functions ::**

predefined : The predefined functions are functions that are built-in, already embedded in the program

User-defined : the user-defined function which functions that a user inputs into the program to perform custom tasks.

**Function methods::**

**Call:** **Call** is a function that helps you change the context of the invoking function.

it helps you to replace the value of (.this) inside a function with whatever value you want.

**Apply:** **Apply** is very similar to the Call function. The only difference is that in Apply you can pass an array as an argument list.

**Bind: Bind** is a function that helps you create another function that you can execute later with the new context of this that is provided.

**There are following ways to create an object in JavaScript ::**

using object literals,

using the function constructor,

//……using the Object.create method,

//…….using the class keyword (which is almost the same as using a function constructor).

**prototype Inheritance::** In JavaScript, an object can inherit properties of another object. The object from where the properties are inherited is called the prototype.

In short, objects can inherit properties from other objects.

**Type Script ::** a programming language developed and maintained by Microsoft. It is becoming increasingly popular.

**Array Methods ::**

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

## Map Method: The map method executes a provided function once for every element in the array and, it **returns a new transformed array.**

## find Method: The find method returns the value of the first element in the array that satisfies the provided test condition.

## findIndex Method: The findIndex method returns the **index** of the first element in the array **that satisfies the provided test condition**. Otherwise, it returns -1, indicating that no element passed the test.

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

## every Method: The every method tests whether all elements in the array pass the provided test conditions and returns a boolean true or false value.

## some Method: The some method tests whether at least one element in the array passes the test condition given by the provided function and returns a boolean true or false value.

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

**String Methods::**

## substring( ) Method: This method basically gets a part of the original string and returns it as a new string.

## The substring method expects two parameters:

* **startIndex**: represents the starting point of the substring
* **endIndex**: represents the ending point of the substring (optional)

EX:

const myString = "I am learning JavaScript and it is cool!";

string.substring(0, 10);

OUTPUT: I am learn

const myString = "I am learning JavaScript and it is cool!";

string.substring(5);

OUTPUT: learning JavaScript and it is cool!

## slice( ) Method: The slice( ) method is similar to the substring( ) method and it also returns a substring of the original string.

## The slice method expects two parameters:

* **startIndex**: represents the starting point of the substring
* **endIndex**: represents the ending point of the substring (optional)

EX:

 const slice = "I am learning JavaScript and it is cool!";

string.slice(14);

OUTPUT: JavaScript and it is cool!

const slice = "I am learning JavaScript and it is cool!";

string.slice(5, 25);

OUTPUT: learning JavaScript

const slice = "I am learning JavaScript and it is cool!";

string.slice(-5);

OUTPUT: cool!

## substr( ) Method:: The substr( ) method also returns a substring of the original string and expects two parameters as:

**startIndex**: represents the starting point of the substring

**length**: number of characters to be included (optional)

EX:

 const myString = "I am learning JavaScript and it is cool!";

string.substr(5,5);

OUTPUT: learn

const myString = "I am learning JavaScript and it is cool!";

string.substr(5);

OUTPUT: learning JavaScript and it is cool!

**Difference Between slice and splice():: splice()** changes the original array.

var array=[1,2,3,4,5];

console.log(array.splice(2));

**slice()** doesn't but both of them returns array object.

var array=[1,2,3,4,5]

console.log(array.slice(2));

**Difference‌ ‌between‌ ‌Java ‌and‌ ‌JavaScript::**

|  |  |
| --- | --- |
| **Java** | **JavaScript** |
| This is OOP or Object-Oriented programming language | This is an object based scripting language(SINGLE THREADED) |
| A stand-alone language | Not stand-alone, incorporated into HTML program for operation |
| Strongly typed language is used, and data type of variable is decided before declaring or using it | Language utilised is loosely typed, so that the user does not have to worry about the data type before the declaration |
| Code has to be compiled  | The code is all text |
| Slightly more complex | Easier in comparison |
| Used to perform complex tasks | Complex tasks cannot be executed |
| Large amount of memory is required | Memory consumption is lesser |
| Programs are saved with “.java” extension | Programs are saved with JavaScript, “.js” extension |
| Stored in the / client host machine under the “Byte” code | Stored in host or client machine as “source” code |
| Compiled on the server before it is executed on the client side | JavaScript is interpreted on the client side |
| Is static and the code once written can be run on any computing platform | Dynamic and is a cross-platform language |

**Throttling** :: this will delay executing a function. It will reduce the notifications of an event that fires multiple times.

**Debouncing** :: This will bunch a series of sequential calls to a function into a single call to that function. It ensures that one notification is made for an event that fires multiple times.

**Undefined ::** variable or anything without a value will always return "undefined" in JavaScript

**Null ::** a primitive type intentionally containing the value of null.

**falsy values ::** A falsy value is something which evaluates to FALSE, for instance when checking a variable. There are six falsey values in JavaScript:

1. undefined, (2) null, (3) NaN, (4) 0, (5) "" (empty string), (6) false.

**Execution Context ::** contains the code that's currently running, and everything that aids in its execution.

During the Execution Context run-time, the specific code gets parsed by a parser, the variables and functions are stored in memory, executable byte-code gets generated, and the code gets executed.

**Event loop ::**

JavaScript has a runtime model based on an **event loop**, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks. This model is quite different from models in other languages like C and Java.

**Stack ::** A very useful data structure and has a wide range of application. Stack is a linear data structure in which addition or removal of element follows a particular order i.e. LIFO(Last in First Out) AND

FILO(First in Last Out).

**queueMicrotask() ::** The **queueMicrotask()** method, which is exposed on the Window or Worker interface, queues a microtask to be executed at a safe time prior to control returning to the browser's event loop.

The microtask is a short function which will run after the current task has completed its work and when there is no other code waiting to be run before control of the execution context is returned to the browser's event loop.

**setTimeout ::** setTimeout() is used to delay the execution of the passed function by a specified amount of time.

There are two parameters that you pass to setTimeout(): the function you want to call, and the amount of time in milliseconds to delay the execution of the function.

setTimeout() will execute the function from the first argument one time after the specified time has elapsed.

**setInterval ::** Use setInterval() to specify a function to repeat with a time delay between executions.

Again, two parameters are passed to setInterval(): the function you want to call, and the amount of time in milliseconds to delay each call of the function .

setInterval() will continue to execute until it is cleared.

**Object.seal() ::** The Object.seal() method seals an object, preventing new properties from being added to it and marking all existing properties as non-configurable. Values of present properties can still be changed as long as they are writable.

**Object.freeze() ::** The Object.freeze() method **freezes** an object. A frozen object can no longer be changed; freezing an object prevents new properties from being added to it, existing properties from being removed, prevents changing the enumerability, configurability, or writability of existing properties, and prevents the values of existing properties from being changed. In addition, freezing an object also prevents its prototype from being changed. freeze() returns the same object that was passed in.

**Map() ::** [Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map) is a collection of keyed data items, just like an Object. But the main difference is that Map allows keys of any type.

* new Map() – creates the map.
* map.set(key, value) – stores the value by the key.
* map.get(key) – returns the value by the key, undefined if key doesn’t exist in map.
* map.has(key) – returns true if the key exists, false otherwise.
* map.delete(key) – removes the value by the key.
* map.clear() – removes everything from the map.
* map.size – returns the current element count.

**Set() ::** Set is a special type collection – “set of values” (without keys), where each value may occur only once.

* new Set(iterable) – creates the set, and if an iterable object is provided (usually an array), copies values from it into the set.
* set.add(value) – adds a value, returns the set itself.
* set.delete(value) – removes the value, returns true if value existed at the moment of the call, otherwise false.
* set.has(value) – returns true if the value exists in the set, otherwise false.
* set.clear() – removes everything from the set.
* set.size – is the elements count.

**sessionStorage ::** object let you store key/value pairs in the browser.

Store data for only one session.(data is deleted when browser is closed)

**localStorage ::** object allows you to save key/value pairs in the browser.

Stores data with no expiration date. Data is not deleted when browser is closed (avilable for future uses.)

**Cookies ::** are data, stored in small text files, on your computer.

When a web server has sent a web page to a browser, the connection is shut down, and the server forgets everything about the user.

Cookies were invented to solve the problem "how to remember information about the user":

When a user visits a web page, his/her name can be stored in a cookie.

Next time the user visits the page, the cookie "remembers" his/her name.

**Write a program to sort an array in javascript::**

let city = ["California", "Barcelona", "Paris", "Kathmandu"];

// sort the city array in ascending order

let sortedArray = city.sort();

console.log(sortedArray);

# JSON.stringify() :: A common use of JSON is to exchange data to/from a web server.

# When sending data to a web server, the data has to be a string.

Convert a JavaScript object into a string with JSON.stringify().

# JSON.parse() :: A common use of JSON is to exchange data to/from a web server.

When receiving data from a web server, the data is always a string.

Parse the data with JSON.parse(), and the data becomes a JavaScript object.

**Generator-Function ::** A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. The yield statement suspends function’s execution and sends a value back to caller, but retains enough state to enable function to resume where it is left off. When resumed, the function continues execution immediately after the last yield run.

**Syntax :**
 // An example of generator function

function\* gen(){

 yield 1;

 yield 2;

 ...

 ...

}

**Generator-Object ::**Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in a “for of” loop. The Generator object is returned by a generating function and it conforms to both the iterable protocol and the iterator protocol.

**stopPropagation() ::** this method prevents propagation of the same event from being called. Propagation means bubbling up to parent elements or capturing down to child elements

**Temporal dead zone** :: Accessing let and  const values before they are initialized can cause a ReferenceError because of the temporal dead zone.

**Currying ::** Currying is a transformation of functions that translates a function from callable as f(a, b, c) into callable as f(a)(b)(c).

# MutationObserver :: The [MutationObserver](https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver) interface provides the ability to watch for changes being made to the DOM tree.

It is designed as a replacement for the older Mutation Events feature, which was part of the DOM3 Events specification.

# Memoization :: **memoization is an optimization technique** that makes applications more efficient and hence faster.

It does this by storing computation results in cache, and retrieving that same information from the cache the next time it's needed instead of computing it again.

**Closures ::** Js variables can belong to the local or global scope. Global variables can be made local(private) with closures.

 A function can access all variables defind inside the function.

A closure is a function having access to the parent scope, even after the parent function has closed.