HTML - content of a webpage
CSS - for styling
Bootstrap - Responsiveness + pre-defined components-table,carousel,modal,card
Javascript - behaviour of a webpage

Javascript
==========
-dynamically typed , Synchronous & single-threaded Programming Language.
-dynamically typed means the types are checked, and datatype mismatches are spotted only at the runtime.
-JavaScript engine executes code from top to bottom, line by line.In other words, it is synchronous.
-JavaScript engine has only one 'call-stack' so that it only can do one thing at a time.

-A JavaScript engine is a program or an interpreter which executes JavaScript code.

javascript Engines:
  V8 - is developed by Google , Google Chrome.
  SpiderMonkey - is developed by Mozilla , Firefox.
  JavaScriptCore - is Apple's engine for its Safari browser.
  Chakra - is the JavaScript engine of the Internet Explorer browser.

Types
------
1. internal
  a. <script></script>
  b. head or body
  c. page level javascript

2. external
  a. separate js file(needs to be included to html)
    <script src='abc.js'></script>
  b. head or body
  c. Application level/project level

Q. What is the best place to include js files?
  head/body?

Eliminate Render-Blocking JavaScript
====================================
-With HTML5, we got two new boolean attributes for the <script> tag : async and defer.
 ex:- <script src='demo.js' async/defer></script>
-async/defer attribute should only be used on external scripts, not with internal scripts.
-These attributes only make sense while using the script in the head portion of the page.
-async attribute ensures that the JavaScript file is loaded asynchronously in the background and does not block rendering.
-defer attribute tells the browser to run the script after the document has been parsed
-With async, the file gets downloaded asynchronously and then executed as soon as it’s downloaded.
-With defer, the file gets downloaded asynchronously, but executed only when the

document parsing is completed.
-With defer, scripts will execute in the same order as they are called.
 defer is useful when a script depends on another script.
 https://flaviocopes.com/javascript-async-defer/

Javascript - is a programming language, follows ECMASCRIPT Standards.
ECMA - European Computer Manufacturers Association.
ECMASCRIPT - is a standard for the scripting languages.

Javascript	TypeScript
1.strongly typed-No	1.strongly typed-yes
2.directly run on browser	2.not directly run on the browser
3.interface-No	3.interface - Yes
4.optional parameters-No	4.optional parameters-yes
5.interpreted language	5.compiles the code
6.errors at runtime	6.errors during the development time
7.generics-no	7.generics-yes

datatypes:
=========
-Types of data we are dealing with.
-JavaScript provides different data types to hold different types of values.
-two types of data types :
      1.primitive   (number,string,boolean,undefined,null,symbol,bigInt)
      2.Non-primitive (object references - Object)
-difference between primitives and non-primitives is that primitives are immutable and
 non-primitives are mutable.

```
      var str = 'This is a string.';
      str[1] = 'H'
      console.log(str); // 'This is a string.'
```

-Mutable values are those which can be modified after creation.
-Immutable values are those which cannot be modified after creation.
-primitives are compared by values whereas non-primitives are compared by references.
-primitive value will always be exactly equal to another primitive with an equivalent value.

```
    const first = "abc" + "def";
      const second = "ab" + "cd" + "ef";
      console.log(first === second); // true
```

-equivalent non-primitive values will not result in values which are exactly equal.

```
    const obj1 = { name: "sanjay" };
      const obj2 = { name: "sanjay" };
      console.log(obj1 === obj2); // false
```

-'typeof' is used to check datatype of a value.

Primitive	Non-primitive/Complex
1. number	1. Object (Object & Array)
2. string	
3. boolean(true,false)	
4. undefined(undefined)	
5. null	

6. symbol(ES-6)
7. bigint (const x = 2n ** 53n;)
   appending n to the end of an integer literal

null vs undefined

-undefined means "not initialized".it means a variable is declared but not initialized yet.
-null means "the intentional absence of any object value". null is an assigned value. It means nothing.
-A variable initialized with undefined means that the variable has no value or object assigned to it
 while null means that the variable has been set to an object whose value is undefined.
-Both null and undefined are falsy values.
-The JSON data format does not support undefined, supports only null;
-JSON.stringify({a: undefined, b: null}) -->{"b":null}
-even though typeof null === 'object', null is a still primitive value.

```
      ex: null == null;   //true
```

1. when a variable is declared without a value, that variable by default will have undefined value
   a variable can be declared with a null value.

```
      var a;                  //a=undefined
      var b = null;  //b=null
```

2. typeof undefined; // undefined
   typeof null; //object

```
 var a = {}
 var b = []
 typeof a; //object
 typeof b; //object

 a instanceof Object; //true
 b instanceof Object; //true

 a instanceof Array; //false
 b instanceof Array; //true

 Array.isArray(a); //false
 Array.isArray(b); //true
```

Symbol
======
-A 'symbol' represents a unique identifier.
-Symbols are often used to add unique property keys to an object that won’t collide with keys.
-create a symbol by calling the Symbol(), not by using new keyword.

```
 let sym1 = Symbol()  // correct
 let sym2 = new Symbol()  // TypeError
```

-Even if we create many symbols with the same description, they are different values.

```
 Symbol('foo') === Symbol('foo')  // false
```

-Symbols allow us to create “hidden” properties of an object,
 that no other part of code can accidentally access or overwrite.
-If we want to use a symbol in an object literal,we need square brackets around it.

```
 let id = Symbol('User Id');
 let user = {name:'sanjay',[id]:123}
-Symbols are not enumerated,Symbols are skipped by for…in while we iterate
object properties.
-Symbols are not part of the Object.keys() or Object.getOwnPropertyNames()
-Symbols assigned to an object can be accessed using the
Object.getOwnPropertySymbols() method
-Object.assign() copies both string and symbol properties.
```

```
Type coercion/Type casting
==========================
-Type coercion is the automatic or implicit conversion of values from one data
type to another (such as strings to numbers).
-Type conversion is similar to type coercion because they both convert values
from one data type to another with one key difference
-type coercion is implicit whereas type conversion can be either implicit or
explicit.

Rules:
-Primitives are coerced to string when using the binary + operator —
  if any operand is a string
-Primitives are coerced to number when using comparison operators.
      "5" >= 1   true
-Primitives are coerced to number when using arithmetic operators (except for +
when one operand is a string)
      "5" - 1    4
      "5" * 3    15
-Primitives are coerced to number when using the unary + operator
      +'123'   123
      +true  1
      +null  0
```

```
Most Used methods in JS
=======================
window.alert()  -  window(Object), alert(function)
document.write() - document(object), write(function)
console.log()   -  console(object), log(function)

window - alert(),confirm(),prompt(),setTimeout(),setInterval(),print(),open()
document - write(),writeln(),getElementById(),querySelector()
console - log(),error(),warn(),table(),dir(),trace(),time(),timeEnd(),group()


N.p - while calling window object functions it is not necessary to call with
object name;
      window.alert();  // correct
      alert();           // correct
      this.alert();    // correct

      console.log() // Yes
      log() // No

      document.write(); //yes
      write(); //No

Note:- all window Object functions can be directly called.
       default value of 'this' is 'window'
```

How Javascript Works

-When a javascript code is executed, N # of execution contexts are created.
-1 Global Execution context is created for every javascript program.
-For Each Function call 1 execution context is created.
-Execution context is the environment within which the current code is being executed.
-Each execution context has two phases:
      1. creation phase. (Allocates memory - variables & functions and assign 'undefined' to variables)
      2. execution phase. (Code gets Executed - assign values to variables & method invocation)

https://www.youtube.com/watch?v=iLWTnMzWtj4&t=1044s
https://www.jsv9000.app/

Variables

-Variables are containers for storing data values.
 Variable is a name of memory location.
-Variables in Javascript can be declared by using either one of the below 3 keywords:
 1. var
 2. let
 3. const

var	let	const
1.since begining	1.ECMASCRIPT-6(2015)	1.ECMASCRIPT-6(2015)
2.value can be changed	2.value can be changed	2.cann't be changed
3.initialization is not mandatory	3.initialization is not mandatory	3.mandatory
4.can be redeclared	4.cann't be redeclared	4.cann't be redeclared
5.TDZ - No	5.TDZ - Yes	5.TDZ - Yes
6.function/global	6.block/function/global	6.block/function/global

N:p - All variables (var,let,const) are hoisted but only 'var' variables are usable/reachable before initialization.
-let/const variables are not reachable/usable before initialization (Temporal Dead Zone)

Hoisting:

-The process of assigning variable declarations a default value of 'undefined' during the creation phase is called Hoisting.
-In hoisting, the variable and function declarations are put into memory during the   compile/creation phase before code execution phase.

Temporal Dead Zone

-The period between entering scope and being declared where they cannot be accessed.
 This period is the temporal dead zone (TDZ).

-The state where variables are un-reachable. They are in memory, but they aren't usable.
-The let and const variables exist in the TDZ from the start of their enclosing scope until they are initialized.
-if a let/const variable is accessed before its declaration, it would throw a ReferenceError. Because of the TDZ.

What’s the difference between context and scope?
===
-The context is (roughly) the object that calls the function.
-And the scope is all the variables visible to a function where it is defined.
-One cares about how it is called, the other cares about how it is defined.

variable scope:
==============
-Scope is a certain section/region of the program where a defined variable can have its existence and can be recognized, beyond that it can’t be accessed.
-Scope determines the visibility and accessibility of a variable.
-Every variable will have either 1 of the below 3 scopes.

1. global
2. function/Local
3. block

global scope:

-variables declared outside function.
-these are accesible/visible throughout the script by any function.

function:

-declared inside a function/function arguements.
-can be used only inside that function.

block scope:

-declared inside a block(if,else,try,catch)
-visible only inside a block

N.P-Scope of the variables declared without var/let/const become global irrespective of
where it is declared.

Scope chain
===========
-While resolving a variable, the block first tries to find it within the own scope.
-If the variable cannot be found in its own scope it will climb up the scope chain and look for the variable name in the environment where the function was defined.
-If the variable cannot be found there, it will climb up the scope chain and will go till global scope to resolve the variable.

Variable Shadowing
==================
-when a variable is declared in a certain scope having the same name defined on its
outer scope and when we call the variable from the inner scope, the value

```
assigned to
 the variable in the inner scope is the value that will be stored in the
variable in
 the memory space. This is known as Shadowing.

-while shadowing a variable, it should not cross the boundary of the scope, i.e.
we can
 shadow var variable by let variable but cannot do the opposite. So, if we try
to shadow
 let/const variable by var variable, it is known as Illegal Shadowing and it
will give the error as “variable is already defined.”



Use Strict
==========
-provides better coding standard and stronger error checking.
-'use strict' is only recognized at the beginning of a script or a function.
-JavaScript modules are automatically in strict mode, with no statement needed
to initiate it.(import/export)
-The purpose of "use strict" is to execute the javascript in "strict mode".
-when 'use strict' is not written, browser runs the JS in normal mode.
-when 'use strict' is  written, browser runs the JS in strict mode.
-Strict mode changes some previously-accepted mistakes into errors.


1. variable declaration without var/let/const is not allowed.
   makes it impossible to accidentally create global variables.
2. function with duplicate arguements are not allowed.
3. NaN/undefined/Infinity cann't be used as a variable name.
4. Delete of an unqualified identifier in strict mode.
    ex:delete Object.prototype;
      var x = 5; delete x;
5. Multiple assignments not allowed.
   var a = b = c = 3;
6. 'this' is undefined, when a function is invoked from Global Context in strict
mode.



Comments
========
-it improves readability/understandability of a file.
-single line comment
// single line comment   (ctrl + /)

-multi line comment    ( Alt + Shift + a)
/*
line-1
line-2
line-3
*/




Operators
=========
1. arithmetic (+,-,*,/,%,**) /quotient %remainder **exponent
2. Assignment (=,+=,-=,*=)
3. Relational  (>,>=,<,<=,==,===,!=,!==)
4. logical (&&,||,!)
5. bitwise  (&,|,^,~)
6. increment/decrement (++,--)
```

```
7. miscelaneous (typeof,instanceof)


No of operands
---------------
1. unary (1 operand)      ex: -5 , ++a
2. binary (2 operands)    ex: a+b , x-y
3. ternary (3 operands)   ex: a>b ? a : b;

res = condition ? trueValue : falseValue;


== compares only the value, performs implicit type conversion. (equality)
=== compares both value and datatype, no type conversion is performed.(Strict
equality)

Note: === (Strict equality) is faster as no type conversion is performed


pre-increment(++a)
post-increment(a++)

pre-decrement(--a)
post-decrement(a--)

Q. using ternary operator find the greatest number amongst 3 numbers



dialog boxes/popup boxes
========================
the below functions are from 'window' object
1.  alert() - display Message
                   (message + ok)
2.  confirm()- User Confirmation
                    ( message + ok-true,cancel-false)
3.  prompt()- Collect User Input
                    (message + inputBox + ok,cancel)




conditional statements
======================
1. if
2. if-else  (nested if-else)
3. switch




Loop
====
Loop helps to execute a block of statements/code number of times.

1. while
2. do-while (usefule to output some sort of menu, the menu is guaranted to show
once)
3. for



Functions
=========
-function is a block of code/statements designed to perform a particular task.
```

-function is executed only when that gets invoked/called.
-function is defined with the function keyword, followed by a name, followed by parentheses ().
-The code to be executed(function body), by the function, is placed inside curly brackets: {}
-Function parameters are listed inside the parentheses () in the function definition.
-Function arguments are the values received by the function when it is invoked.
-Inside the function, the arguments (the parameters) behave as local variables.

1. pre-defined (alert(),prompt(),confirm(),max(),min(),sqrt(),cbrt())
   already written, we are just using them
2. user-defined
   we have to write,and we will use them

   a. function declaration (named function)
   b. function expression(anonymous)
   c. self invoked (IIFE- Immediately Invoked Function Expression)
   d. arrow function (ES - 6)(2015)

Note:

Parameter: is the variable in the function declaration.
It is part of the function signature when you create it.
Argument: is the actual value of the variable being passed to the function when it is called.

Function Declaration	Function Expression
1. Named	1. Anonymous
2. Hoisting - yes	2. Hoisting - No
3. creation phase(parse)	3. execution phase (run)

-A Function Expression is created when the execution reaches it and is usable only from that moment.
-A Function Declaration can be called before it is defined.
-function declarations are parsed before their execution.
function expressions are parsed only when the script engine encounters it during execution.

Arrow Function

-'this' value inside a regular function is dynamic and depends on the context in which it is called.
-'this' inside the arrow function is bound lexically and equals to 'this' where the function is declared.
-lexical context means that arrow function uses 'this' from the code that contains the arrow function.
-Regular function (this = how the function is invoked/who invoked)
-Arrow function(this = where the function is declared)

Arrow Function Limitations

-Arrow functions don't have their own bindings to this, arguments or super cann't be used inside arrow function.
-Arrow functions don't have access to the new,target keyword.
-Arrow functions aren't suitable for call, apply and bind methods, which generally rely on establishing a scope.

-Arrow functions cannot be used as constructors.
-Arrow functions cannot use yield, within its body.

IIFE
====
-used when we try to avoid polluting the global scope,
-The variables declared inside the IIFE are not visible outside its scope.
-closure

Function Curring
================
-Function Currying is a concept of breaking a function with many arguments into many functions with single argument in such a way, that the output is same.
-its a technique of simplifying a multi-valued argument function into single-valued argument multi-functions.
-It helps to create a higher-order function. It is extremely helpful in event handling.

```
      var add = function (a){
                 return function(b){
                       return function(c){
                                 return a+b+c;
                              }
                        }
                  }
      console.log(add(2)(3)(4)); //output 9
```

Pure Function
=============
-Pure functions are functions that accept an input and returns a value without modifying any data outside its scope(Side Effects).

-A function is called pure if that follows the below 3 standards
      1. Pure functions shouldn't update the data outside it's scope.
      2. pure functions must return a value.
      3. Its output or return value must depend on the input/arguments.

Higher-order Function
=====================
-Higher-order function is a function that may receive a function as an argument and/or can even return a function.
-a function can be called as a Higher-order if that function has either of the below 2 abilities:
      1. a function has ability to return another function.
      2. a function has ability to take another function as argument.
-Array filter(),map(),reduce(),sort() are some of the Higher-Order functions.

function recursion
==================
-A recursive function is a function that calls itself until the program achieves the desired result.
-A recursive function should have a condition that stops the function from calling itself.otherwise, 'RangeError: Maximum call stack size exceeded' error

will be thrown
-A recursive function can be used instead of a loop where we don't know how many times the loop needs to be executed.

```
ex: function countDown(fromNumber) {
            console.log(fromNumber);
            let nextNumber = fromNumber - 1;
            if (nextNumber > 0) {
                  countDown(nextNumber);
            }
      }
      countDown(5);
```

Memoization
===========
-Memoization is a programming technique that attempts to increase a function’s performance by caching its previously computed results.
-Memoization is an optimization technique used to speed up performance by storing the results of expensive function calls and returning the cached result when the same inputs occur again.
-its a kind of caching the data.
-used with recursion.

function closure
================
-A closure is an inner function that has access to its outer function's variables even after the outer function's execution is Completed/Closed.

-When the outer function execution completes, you’d expect all its variables to be no longer accessible. However, if the inner function uses variables from the outer function, those variables remain accessible.

-The inner function retains access to the outer function’s scope, because the inner function ‘remembers’ the environment in which it was created.

Closure Disadvantages
=====================
-As long as the closure are active , the memory can't be garbage collected.
-If we are using closure in ten places then unless all the ten process complete
 it hold the memory which cause memory leak.

Number
======
isInteger();
isNaN()
parseInt();
parseFloat();

Math
=====
Math.PI;
abs();

```
sqrt();
cbrt();
ceil();
floor();
round();
max();
min();
pow();
random();  // 0.0 - 1.0



Strings
=======
-strings are used to represent and manipulate a sequence of characters.
-JavaScript string is zero or more characters written inside quotes.
-We can use single or double quotes for string.
        var a='hello';
        var b="hello";
-We can use quotes inside a string, as long as they don't match the quotes
surrounding the string.
      var answer1 = 'It's alright'; //in-valid
      var answer1 = "It's alright"; //valid
      var answer2 = "He is called 'Johnny'";  //valid
      var answer3 = 'He is called "Johnny"'; // valid
-Strings can be created in 2 ways
      1. as primitives, using string literals;
         var a = 'hello';
      2. as objects, using the String() constructor
         var b = new String('hello');
-JavaScript automatically converts primitives to String objects, so that it's
possible
 to use String object methods for primitive strings.
-String primitives and String objects give different results when using eval().
 Primitives passed to eval are treated as source code; String objects are
treated as all other objects are, by returning the object..
      let s1 = '2 + 2'              // creates a string primitive
      let s2 = new String('2 + 2')  // creates a String object
      console.log(eval(s1))         // returns the number 4
      console.log(eval(s2))         // returns the string "2 + 2"
-A String object can always be converted to its primitive counterpart with the
valueOf() method.
      console.log(eval(s2.valueOf()))  // returns the number 4


1. literal
   var str1 = "sachin";
   typeof str1; //string

2. object
   var str2 = new String("sachin");
   typeof str2;   //"object"

var a = "sachin";
var b = "sachin"
a == b; // true

var a = "sachin";
var b = new String("sachin")
a == b; // true

var a = new String("sachin");
var b = new String("sachin");
a == b;  // false
```

```
String methods:
--------------
length;
toUpperCase();
toLowerCase();
charAt();
at()
charCodeAt();
concat();
indexOf();
lastIndexOf();
includes();
match();
matchAll()   // returns iterator
replace();
replaceAll();
slice(start, end)
substring(start, end)
substr(start, length)
split()
search(regex)
trim()
eval(); (eval() is from Window Object)
localeCompare()

N.p:- substring() cannot accept negative indexes. slice() does.



Array
=====
-Arrays are used to store multiple values in a single variable.
      ex: var arr = [10,20,30,40,50]
-An array can hold many values under a single name, and we can access the values
by referring to an index number.
      ex: console.log(arr[1]);
-Usually in other programming languages array stores similar type of
elements,but in
 JavaScript array can have heterogeneous elements.
      ex: var arr = [10,'sachin',true,{}]


array creation:
---------------
var arr1 = [10,20,30,40,50];
var arr2 = new Array(5);
var arr3 = new Array(10,20,30,40,50);

-iterating over an array: 1.loop   2.for-in   3.for-of      4.forEach()

properties: length , delete

instance functions:
at(),concat(),entries(),every(),fill(),filter(),find(),findLast(),findIndex(),fl
at(),
flatMap(),forEach(),includes(),indexOf(),join(),keys(),lastIndexOf(),map(),pop()
,push()
reduce(), reverse(), shift(),slice(),sort(),some(), splice(),unshift(),values()

static functions : from(),isArray(),of()
```

To add/remove elements:

push(...items) – adds items to the end,
pop() – extracts an item from the end,
shift() – extracts an item from the beginning,
unshift(...items) – adds items to the beginning.
splice(pos, deleteCount, ...items) – at index pos delete deleteCount elements and insert items.
toSpliced()   - doesn't change the original array
slice(start, end) – creates a new array, copies elements from position start till end (not inclusive) into it.
concat(...items) – returns a new array: copies all members of the current one and adds items to it. If any of items is an array, then its elements are taken.
with(ind,newValue) - Create a new array with a single element changed

To search among elements:

at(index) - takes an integer and returns the item at that index.allows negative index aswell.
indexOf/lastIndexOf(item, pos) – look for item starting from position pos, return the index or -1 if not found.
includes(value) – returns true if the array has value, otherwise false.
find(func) – filter element through the function, return first value that make it return true.
findLast(func) – filter element through the function, return last value that make it return true.
filter(func) – filter elements through the function, return all values that make it return true.
findIndex(func) - it is like find(), but returns the index instead of a value.

To transform the array:

map(func) – creates a new array from original array by calling func for every element.
sort(func) – sorts the array in-place, then returns it.
toSorted() - creates a new array and sorts
reverse   () – reverses the array in-place, then returns it.
toReversed() - creates a new array and reverses
split/join – convert a string to array and back.
reduce(func, initial) – calculate a single value over the array by calling func for   each element and passing an intermediate result between the calls.
flat()   - creates a new array with the elements of the subarrays concatenated into it.flat(Infinity) , flat() also removes holes in array
flatMap() - maps each element in an array using a mapping function and then flattens the results into a new array

To iterate over elements:

forEach()   – calls func for every element, does not return anything.

Additionally:

Array.isArray(arr) checks arr for being an array.
Array.from() change array-like or iterable into true array
Array.of() create array from every arguments passed into it.

```
const nums = Array.of(1, 2, 3, 4, 5, 6);
console.log(nums);


let mySet = new Set()
mySet.add(2).add(3).add(4);
console.log(Array.from(mySet))

const lis = document.querySelectorAll('li');
const lisArray = Array.from(document.querySelectorAll('li'));

// is true array?
console.log(Array.isArray(lis)); // output: false
console.log(Array.isArray(lisArray));  // output: true

var arr = [10,20,30]
var x = arr.values();
console.log(x)
var y = Array.from(x)
console.log(y)
```

Array Copy

==========

```
let arr1 = [10, 20];
let arr2 = arr1;  // address/reference copy ( Not value copy)
```
-A new array is not being created, rather same address is being assigned to arr2
-Both arr1 & arr2 are holding the same address

```
let arr1 = [10, 20, [30, 31]];
let arr2 = [...arr1]; // value copy - shallow copy

let arr3 = [10, 20, [30, 31]];
let arr4 = structuredClone(arr3); // value copy - Deep Copy
```

-A shallow copy of an array is a copy whose nested elements share the same references.
 (Nested arrays will not be copied by value)
-A deep copy of an array is a copy whose nested elements do not share the same references.

OOP

```
class - structure/blueprint/template for creating Object
            class has only logical existance
            class doesn't have physical existance

object - Real Entity
             every instance of a class
             Object has physical existance
```

-a class in javascript is created using constructor function(ES-5).
-a class in javascript is created using class keyword.(ES-6).
-class contains variables(states/properties) and methods(behaviours) inside it.

Prototype

-A prototype is an object used to implement structure, state, and behaviour inheritance.
-Prototype is the mechanism by which JavaScript objects inherit features from one another
-Prototype is an object, where we can attach methods and properties,which enables all the other objects to inherit these methods and properties.
-Prototype is a base class for all the objects, and it helps us to achieve inheritance.
-All JavaScript objects inherit properties and methods from a prototype.
-properties added to the prototype of a class gets available to all the objects of that class.

-prototype should be used When we have to add new properties like variables and methods
at a later point of time,and these properties needs to be shared across all the instances,
-a property should be added to the constructor of a class if value of the property changes per object
-a property should be added to the prototype of a class if the value remains same for all objects.

Inheritance

- Inheritance is the concept where one class inherits the properties
  from another class.
- the class which inherits properties is called child/derived/sub.
  the class which provides the properties is called parent/base/super.
- it is mainly used for code re-usability.
- also called is-a relationship

'Object' class
==============
-Objects are variables that can contain many values inside it.
-Collection of properties & values.
   ex: {prop1:val1,prop2:val2}
-Object properties are written in 'key:value' pairs.
  ex: let user = {name:'sachin' , age:35, add:'mumbai'}
-we can access object properties in two ways:
```
      objectName.propertyName;       user.name;
      objectName["propertyName"];    user['age'];
```
-4 ways to create javascript object
```
   1. Object Literal    ex: var obj1 = {};
   2. Object create()   ex: var obj2 = Object.create({});
   3. Object Class          ex: var obj3 = new Object();
   4. Using Class       ex: var obj4 = new Employee();
```
-How to get the length of the object
```
      Object.keys(obj).length;
```

-How to check if a property exists in an object
```
 console.log(propertyName in obj)   (also includes prototype properties)
 obj.hasOwnProperty(propertyName)   (doesn't includes prototype properties)
```

-Object class static functions :
      assign() - Copies properties from one or more source objects to a target object.
      create() - creates a new object, using an existing object as the prototype of the newly created object
      freeze() - Freezes an object. neither the structure nor values can be changed
      isFrozen() - Determines if an object was frozen
      seal() - structure of the object cann't be modified, value of the properties can be changed.
      isSealed() - Determines if an object is sealed.
      preventExtensions() -new properties cann't be added to an object, properties can be deleted, value of the properties can be changed
    isExtensible() - Determines if extending of an object is allowed
      keys()   - Returns an array of keys
      values() - Returns an array of values
      entries() - returns an array of [key, value] pairs
      fromEntries() - transforms an array/Map into an Object
      groupBy() - groups the data

```
-JSON.stringify()  converts object to string.
-JSON.parse()      converts string to object.


-shallow copy : Object.assign()
                obj2 = {...obj1}
-deep cloning :  JSON.parse(JSON.stringify(obj))
                         obj2 = structuredClone(obj1)
```

Note: structuredClone() cann't copy the functions from original object to copied object.

-A shallow copy of an object is a copy whose nested properties share the same references.
(Nested objects will not be copied by value)
-A deep copy of an object is a copy whose nested properties do not share the same references.

this keyword

-'this' is the context of a function invocation/execution.

-In the global execution context (outside of any function), 'this' refers to the global object whether in strict mode or not.
-'this' is undefined in a function invocation in strict mode.
-'this' is the object that owns the method(not arrow function) in a method invocation.
  the object becomes value of 'this'. myObj.myMethod();
-'this' is the newly created object in a constructor invocation.
-'this' is the first argument of .call() or .apply() in an indirect invocation.
-'this' is the first argument of myFunc.bind(thisArg) when invoking a bound function.
-'this' is the enclosing context where the arrow function is defined.
-Inside a setTimeout function, the value of this is the window object.
-For dom event handler, value of this would be the element that fired the event.

note:- You can always get the global object(window) using 'globalThis' property, regardless of the current context in which your code is running.

Call() & apply() & bind()
========================
-call(),apply(),bind() are methods from 'Object' class.
-used to change the context(this value) while calling a function.
-while calling a function, if we want to pass 'this' explicitely.
-Calls a method of an object, substituting another object for the current object.

call() - call() is used to pass differenet object as a value to 'this'.
call() method calls a function with a given 'this' value and arguments provided individually.
using call() one object can invoke another object's function.

apply() - apply() takes 2 arguments.1st arguement is an object(this), 2nd arguement is an array of items.
-it takes the values from that array and passes as individual arguements to a method.

-bind() creates a new function and when that(new function) is called will have its 'this' set to the provided value with a given sequence of arguements.
-it is most useful for binding the value of 'this' in methods of classes
  that you want to pass into other functions.

```
ES-6 Features (ES-2015)
=======================
1. const & let
2. From IIFEs to blocks
3. concatenating strings to template literals
4. Multi-line strings
6. default parameter values
7. from arguments to rest parameters(...)
8. exponent operator (**)
9.  Desturcturing (array/object)
10. for-loop to for-in and for-of
11. arrow functions
12. From apply() to the spread operator (...)
13. Enhanced Object Literals
14. From constructors to classes
15. inheritance - class,extends
16. Modules
17. collection-Map,weakmap,Set,weakset
18. promise and async-await
19. generators
20. Tail Calls



Modules
=======
-2 Module Systems
      a. CommonJS Module System      (module.export , require())
      b. ECMASCRIPT Module System    (export , import)



Event Handling
--------------
- onclick,ondblclick,onmouseover,onmouseout (Mouse)
- onkeypress , onkeydown, onkeyup  (Keyboard)
- onsubmit, onchange, onblur,onfocus,onpaste,oncopy  (Form)
- onload, onbeforeUnload , onunload (Document)


addEventListner
---------------
1. to add events to dynamically added elements.
2. to add multiple events to an element.


Event Delegation
----------------
-Event delegation allows to add event listeners to one parent instead of
 adding event listeners to many child elements.
-That particular listener analyzes bubbled events to find a match on the child
elements.


Event Propagation:-
1. event Capturing (parent-->child)
2. event Bubbling  (child-->parent)

bubbling:
--------
```

-When an event gets triggered on an element, it first runs the handlers on it, then on
 its parent, then all the way up on other ancestors/parents. (Event Bubbling)
-Event bubbling is a way of event propagation in the HTML DOM API, when an event occurs in an element
 inside another element, and both elements have registered a handle for that event.
 With bubbling, the event is first captured and handled by the innermost element and then propagated to outer elements. The execution starts from that event and goes to its parent element.
 Then the execution passes to its parent element and so on till the body element.

How to stop that?
Answer:-
   stopPropagation() / cancelBubble (IE)

   stopImmediatePropagation (Execute the first event handler, and stop the rest of
the event handlers from being executed)

DOM (Document Object Model)
==========================
-The Hierarchical representation of all html elements inside browser is DOM.
-The DOM represents a document as a tree of nodes.
-The Document Object Model (DOM) is an application programming interface (API) for manipulating HTML documents.
-It provides methods that allows to add, remove, and modify parts of the document effectively.

Uses:

-display content conditionally(After page load)
-add/remove/update content dynamically.
-change css after page load.
-add event listener to the elements which are dynamically added.

Find Elements from DOM
----
1. document.getElementById(id); //Single Element
2. document.getElementsByClassName(className); //Array of elements
3. document.getElementsByTagName(tagName); //Array of elements
4. document.getElementsByName(name); //Array of elements

5. document.querySelector(); //Single Element
6. document.querySelectorAll();//Array of elements

get value from inputBox
--var x = document.getElementById('name').value

set value to an input box
--document.getElementById('name').value = "new value"

- CreateElement Dynamically: createElement(), appendChild()
- attributes : getAttribute(),
setAttribute(),removeAttribute(),toggleAttribute()
- CSS Class : addClass() , removeClass() , toggleClass()

```
innerText vs innerHTML
----------------------
innerText : returns only text contained by an element and all its child
elements.
innerHtml : returns all text, including html tags, that is contained by an
element.



Collections
===========
object vs map
-------------
1. object keys must be strings/symbol, where as map keys can be of any type.
2. You can get the size of a Map easily.  ex: map.size
   for object size has to be calculated manually. ex: Object.keys(obj).length
3.The iteration of maps is in insertion order of the elements
4.Objects are not inherently iterable (Object.keys() is slow), Maps are
iterable.


Map
===
-Maps are collections of keys and values of any type
-const myMap = new Map(); // Create a new Map
 myMap.set('name', 'sanjay'); // Sets a key value pair
 myMap.set('hobby', 'cycling'); // Sets a key value pair
-get data from map
 console.log(myMap.get('name'));
-size of map
 console.log(myMap.size);
-Iterate Map
 for (const [key, value] of map) {
   console.log(`${key} = ${value}`);
 }


Set
===
-Sets are ordered lists of values that contain no duplicates.
-const planetsOrderFromSun = new Set();
 planetsOrderFromSun.add('Mercury');
 planetsOrderFromSun.add('Venus').add('Earth').add('Mars'); // Chainable Method
 console.log(planetsOrderFromSun.size);
 console.log(planetsOrderFromSun.has('Earth')); // True
 planetsOrderFromSun.delete('Earth'); // deletes 1 item
 planetsOrderFromSun.clear(); // deletes all the items


Garbage Collections
-------------------
-JavaScript Garbage Collection is a form of memory management whereby objects
that are no longer referenced are automatically deleted and their resources are
reclaimed.
-Map and Set's references to objects are strongly held and will not allow for
garbage collection.
This can get expensive if maps/sets reference large objects that are no longer
needed, such as DOM elements that have already been removed from the DOM.


WeakMap & Weakset
---------------
-A WeakMap is a collection of key/value pairs whose keys must be objects.
```

-an object's presence as a key in a WeakMap does not prevent the object from being garbage collected.

-WeakSet is Set-like collection that stores only objects and removes them once they become inaccessible by other means.
-Their main advantages are that they have weak reference to objects, so they can easily be removed by garbage collector.
-That comes at the cost of not having support for clear, size, keys, values…
-WeakMap and WeakSet are used as “secondary” data structures in addition to the “primary” object storage.
Once the object is removed from the primary storage, if it is only found as the key of WeakMap or in a WeakSet, it will be cleaned up automatically.

Date
=====

```
var ob = new Date();
ob.getDate();
ob.getMonth();
ob.getYear();
ob.getFullYear();
ob.getTime();
ob.getHours();
ob.getMinutes();
ob.getSeconds();

ob.toLocaleDateString();
ob.toLocaleTimeString();
ob.toLocaleString('en-US', { timeZone: 'America/New_York' })
```

setTimeout/setInterval (Window)
========================
-setTimeout allows to run a function once, after a specified amount of time.
-setInterval allows to run a function repeatedly after the specified interval of time.
-Methods setTimeout(func, delay, ...args) and setInterval(func, delay, ...args) allow us to run the func once/regularly after delay milliseconds.
-To cancel the execution, we should call clearTimeout/clearInterval with the value returned by setTimeout/setInterval.
-Zero delay scheduling with setTimeout(func, 0) (the same as setTimeout(func)) is used to schedule the call “as soon as possible, but after the current script is complete”.
-setTimeout expects a reference to a function, the function shouldn't be invoked.

```
      setTimeout( f1 , 3000);  // correct
      setTimeout( f1() , 3000);  // wrong
```

-For setInterval the function stays in memory until clearInterval is called.
-use recursive setTimeout() insteadof setInterval() if execution duration is longer than
interval time.
-Recursive setTimeout guarantees a delay between the executions, setInterval – does not.
-While this pattern does not guarantee execution on a fixed interval, it does guarantee that
the previous interval has completed before recursing.

```
      (function loop(){
         setTimeout(function() {
              // Your logic here
              loop();
```

```
        }, delay);
      })();


id = setTimeout();
id = setInterval();
clearTimeOut(id);
clearInterval(id);

N.p - setTimeout(fn,0) means execute after all current functions in the present
queue gets executed

https://www.google.co.in/search?
q=javascript+micro+vs+macrotask+queue&tbm=isch&ved=2ahUKEwjF87j61Iz4AhVOkdgFHZd0
BXIQ2-
cCegQIABAA&oq=javascript+micro+vs+macrotask+queue&gs_lcp=CgNpbWcQAzoECCMQJzoFCAA
QgAQ6BggAEB4QCDoECAAQGFCm_BJYh7MTYNq0E2gBcAB4AIAB3QKIAb4kkgEIMC4xMy42LjSYAQCgAQG
qAQtnd3Mtd2l6LWltZ8ABAQ&sclient=img&ei=gpKXYsWqGM6i4t4Pl-
mVkAc&bih=569&biw=1280#imgrc=a3KGFGZQG96R6M
https://www.jsv9000.app/



BOM- Browser Object Model
=========================
-The Browser Object Model (BOM) is used to interact with the browser.

1. window - alert,prompt,confirm,open,close,print
2. screen -
      screen.width
      screen.height
      screen.availWidth
      screen.availHeight

3. Location - window.location
   protocol,host,hostname,pathname,href
   refresh a page  - window.location.reload();
   open a new page - window.location.assign('url');

4. History -
   back(),forward(),length(),go()

5. Navigator -
  The window.navigator object contains information about the visitor's browser.
  navigator.appName
  navigator.appVersion
  navigator.appCodeName
  navigator.platform
  navigator.language
  navigator.vendor
  navigator.userAgent



Cookie vs Offlinestorage
-------------------------
1. cookie- Session id,session token, Visit count,currency,Network-Type
               stores data in browser memory
               text/string data
           name = value pair

---store only limited data(4kb)
---not secure
---with every server request cookie data is sent to server
```

```
---cookie data gets appended to the url and it will be sent

advantage-cookie data can be used @clientside(browser) and
                  @serverside(server)

get Cookie
-----------
var x = document.cookie;
x.split(';')


set Cookie
-----------
document.cookie = "userName='sachin'"

Remove Cookie
--------------
set the expires parameter to a passed date.
-we can delete cookie by setting the cookie to an older date;

-document.cookie ="userName='sachin'; expires=sat, 01 sep 2018 00:00:00 UTC; "

************************

2. offline storage- (HTML-5) (sessionStorage,localStorage)
         stores data in browser
         key-value pair
         text data (object-->JSON.stringify(object)-->string)
                         (string-->JSON.parse(string)-->object)

---can store up to 5 mb
---data will not be sent in the URL
---secure
---data can be used only in browser(clientside/browser)

a. session storage
--session storage data will be there only for a single session.
--data gets lost when we close the tab.

      sessionStorage.length;
      sessionStorage.setItem(key,value);
      value = sessionStorage.getItem(key);
      sessionStorage.removeItem(key);
      sessionStorage.clear();

b. local storage (data remains there even after the tab/window is closed)

N.P - for sensitive information we should use session storage, not local
storage.

-Offline storage can store only String data.
-if data is there in object form, convert to string and store in offline storage
-while reading data, it always returns string, if you expect object then convert
it to object form

store data in Session storage
----------------------------
sessionstorage.setItem(key,value);
OR
sessionstorage.key = value

get data from sessionstorage
----------------------------
var value = sessionStorage.get(key);
```

```
OR
var value = sessionStorage.key;

remove data from sessionstorage
--------------------------------
sessionStorage.removeItem(key);

remove all the items from sessionStorage
---------------------------------------
sessionStorage.clear();




Error Handling
--------------
1. EvalError -
2. RangeError
3. TypeError
4. SyntaxError
5. ReferenceError
6. URIError


when Exception occurs
1. exception object gets created and thrown
2. it is checked whether user handles that exception or not(try-catch)
3. if No, that exception object goes to Default exception handler
   default exception handler-prints exception information and
   stops program execution
4. if yes, exception goes to developer written exception handling code


Terminologies
-------------
1. try- containes a block statements where exception might occur
2. catch - actual exception handling code, this will be executed
         only if exception is there in try
3. finally- contains the statements to be executed at any situation(important
statements)
4. throw- to throw exception explicitely(user-defined exceptions)

N.p: - try block should be immediately followed by either a catch()
       or a finally{}


Form Validation
----------------
1. server side validation
2. client side validation (browser)

ex: inputBox is filled or not
    password format is matching or not


REGEX
------
REGEX - Regular Expression


CallBack
--------
-A Callback is a function that is passed into another function as an argument to
be executed later.
```

-callback function is run inside of the function it was passed into.
-A function that accepts other functions as arguments is called a higher-order function,
 which contains the logic for when the callback function should be executed.

Callback hell
=============
-Callback hell will have multiple callbacks nested after each other.
-It can happen when you do an asynchronous activity that's dependent
 on a previous asynchronous activity.
-Solutions to callback hell:

1. Split functions into smaller functions.
2. Using Promises.
3. Using Async/await.

Promise
=======
-A promise is an object that holds the future value of an asynchronous operation.
-A Promise object represents a value that is not available now, but will be resolved/available at some point in the future.
-Promise object can have different states:- pending, resolved/fulfilled, rejected.
ex:-if we request some data from a server, promise promises us to get that data
 that we can use in the future.

methods of Promise class:

Promise.all(promises) – waits for all promises to resolve and returns an array of their results. If any of the given promises rejects, it becomes the error of Promise.all, and all other results are ignored.
Promise.allSettled(promises) – when all of the given promises have either fulfilled or rejected
      status: "fulfilled" or "rejected"
      value (if fulfilled) or reason (if rejected).
Promise.race(promises) – waits for the first promise to settle, and its result/error becomes the outcome.
Promise.any(promises) – waits for the first promise to fulfill, and its result becomes the outcome.
                        If all of the given promises are rejected, AggregateError becomes the error of Promise.any.
Promise.resolve(value) – makes a resolved promise with the given value.
Promise.reject(error) – makes a rejected promise with the given error.

Async & Await

-async makes a function return a Promise,await makes a function wait for a Promise.
-Async functions enable the programmer to write promise-based code, Syntax and structure of code appears like synchronous code but doesn't block the main thread of execution.
-async and await makes the program very clean and understandable.there is no need of .then()
-we can declare the Async functions in JavaScript by specifying the "async" keyword in front of the function definition.
-For handling the async functions, we use the "await" keyword while invoking to

function to wait for the promise to resolve.

CallStack
=========

-JavaScript engine uses a call stack to manage execution contexts: the Global Execution Context and Function Execution Contexts.
-The call stack works based on the LIFO principle i.e., last-in-first-out.
-When we execute a script,JavaScript engine creates a Global Execution Context and pushes it on top of the call stack.
-Whenever a function is called, the JavaScript engine creates a Function Execution Context for the function, pushes it on top of the Call Stack, and starts executing the function.
-If a function calls another function, the JavaScript engine creates a new Function Execution Context for the function that is being called and pushes it on top of the call stack.
-When the current function completes, the JavaScript engine pops it off the call stack and resumes the execution where it left off in the last code listing.
-The script will stop when the call stack is empty.

-Macro tasks include keyboard events, mouse events, timer events, network events, Html parsing,
changing Url etc. A macro task represents some discrete and independent work.
-Microtasks, are smaller tasks that update the application state and should be executed before
the browser continues with other assignments such as re-rendering the UI.
Microtasks include promise callbacks and DOM mutation changes. Microtasks enable us to execute
certain actions before the UI is re-rendered, thereby avoiding unnecessary UI rendering that
might show an inconsistent application state.

Event Loop

-event loop is a constantly running process that coordinates the tasks between
call stack and callback queue to achieve concurrency.
-event loop monitors both the 'callback queue' and 'call stack'.
-If the callstack/executionstack is not empty, the event loop waits until it is empty and
places the next function from the callback queue to the call stack.
-If the callback queue is empty, nothing will happen.

Stackoverflow Error
===================

-The call stack has a fixed size.
-If the number of execution contexts exceeds the size of the stack, a stack overflow will occur.
```
-ex: function fun1(){
            fun1();
       }
```
-To Solve
```
  function fun1(){
            setTimeout(fun1,0);
      }
```

-The stack overflow is eliminated because the event loop handles the recursion, not the call stack.
-fun1() is pushed to the event queue and the function exits, thereby leaving the call stack clear.

-web browser has other components along with javascript engine.
-setTimeout(), AJAX calls, and DOM events are parts of Web APIs of the web browser.

Debouncing & Throttling

-technique of limiting the number of times the user can call a function attached to an event listener is debouncing and throttling.

-debouncing executes the function only after some cooling period.
-throttling executes the function at a regular interval

-In the debouncing technique, no matter how many times the user fires the event, the attached function will be executed only after the specified time once the user stops firing the event.

-Throttling is a technique in which, no matter how many times the user fires the event, the attached function will be executed only once in a given time interval.

-Debouncing and throttling also prevents the server from being bombarded by the API calls

-Suppose the makeAPICall function takes 500 milliseconds to get data from the API. Now, if the user can type 1 letter per 100 milliseconds, then the makeAPICall function will be called 5 times in 500 milliseconds. Thus even before the makeAPICall has completed its task and returned the response,
 we are making 4 more API calls, which will put extra load on the server

Use of Debouncing and Throttling in Real Life

-We can throttle a button click event, if we do not want the user to spam by clicking the button frequently.
-In the case of window resize event, if we want to redraw the UI only when the user has fixed the size of the window, we can use debouncing.
-With Mouse move event, if we are displaying the location coordinates of the mouse pointer, it is much better to show the coordinates once the user reached the desired location. Here, we can use debouncing

generators

==========

-Generators are a special type of functions that simplify the task of writing iterators.
-produces a sequence of results instead of a single value.
-Generators are functions which can be exited and later re-entered.
-Their context (variable bindings) will be saved across re-entrances.
-When called initially, generator functions do not execute any of their code,
 instead returning a type of iterator called a Generator.
 When a value is consumed by calling the generator's next method,
 the Generator function executes until it encounters the yield keyword.

Javascript Proxy Object

========================

https://melkornemesis.medium.com/javascript-proxy-objects-and-why-you-should-care-with-examples-f9773662e779

Polyfills
=========
-how to make out modern code work on older engines that don’t understand recent features yet?
There are two tools for that:
1.Transpilers
2.Polyfills

-A transpiler can parse (“read and understand”) modern code, and rewrite it using older syntax
constructs, so that the result would be the same.
-Transpilers - Babel,webpack

-Polyfills: A script that updates/adds new functions is called “polyfill”. It “fills in” the gap and adds missing implementations
-libraries of polyfills: coreJS,polyfill.io

Tail Call Optimization
======================
-tail call optimization means that it is possible to call a function from another function without growing the call stack.
-Tail code optimization takes a recursive function and generate an iterative function using “goto” internally

POSTMAN
=======
-Application, used to Test REST APIs. (chrome://apps/)
-Send requests, get responses, and easily debug REST APIs

Fake Online REST API for Testing
----
1. https://jsonplaceholder.typicode.com/
2. https://reqres.in/
3. https://fakestoreapi.com/products

Create REST API with json-server
----
https://medium.com/codingthesmartway-com-blog/create-a-rest-api-with-json-server-36da8680136d

```
1.  Install json-server (not necessarily in a particular folder)
      npm install -g json-server
2. create a json file (anywhere) & paste some data
   users.json (filename can be anything.json)
3. start json server
   json-server --watch users.json
```

```
http://localhost:3000/employees
GET    /employees
GET    /employees/{id}
POST   /employees
PUT    /employees/{id}
PATCH  /employees/{id}
DELETE /employees/{id}
```

```
resources
=========
https://www.w3schools.com/js/
https://javascript.info/
https://developer.mozilla.org/en-US/docs/Web/JavaScript
https://github.com/sudheerj/javascript-interview-questions
http://es6-features.org/#Constants
https://www.jsv9000.app/
https://www.javascripttutorial.net/
```