Variables in JavaScript

- three keywords used to create a variable - **const**, **let** and **var**

Example -

- constant values cannot change

```
const accountID = 144553
let accountEmail = "gaurav@gmail.com"
var accountPassword = "12345"
accountCity = "Nagpur"
```

- cannot print **accountID** because its declared as constant

```
accountID = 2
console.log(accountID);
```

Changing the values of a variables -

Example -

- **console.table()** function prints all the index and values from variable

```
accountEmail = "gm@yahoo.com"
accountPassword = "21011996"
accountCity = "Thane"
console.table([accountID, accountEmail, accountPassword, accountCity])
```

output :

```
┌─────────┬──────────────────┐
│ (index) │ Values           │
├─────────┼──────────────────┤
│ 0       │ 144553           │
│ 1       │ 'gm@yahoo.com'   │
│ 2       │ '21011996'       │
│ 3       │ 'Thane'          │
└─────────┴──────────────────┘
```

- In JavaScript we can use 2 kaywords for the variable - **var** and **let**

- We cannot use **var** keyword due to block scope and functional scope issue
- The value of **accountState** is undefined if not assigned any value to the variable

```
let accountState;
console.log(accountState);
```

output : undefined

Data Types in JavaScript

- treat all JavaScript code as a newer version : **"use strict";**

Primitive Data Types -

- number
- bigint
- string -> ""
- boolean -> true/false
- null -> standalone value (representation of empty values)
- undefined -> value is not defined
- symbol -> unique

Non-Primitive -

- object
- functions
- arrays

Examples of Data Types -

```
let number1 = 10
let number2 = 17456272927334475347484n
let name = "Gaurav"
let isLoggedIn = true
let value = null
let accountState

console.table([number1, number2, name, isLoggedIn, value, accountState])
```

output :

```
┌─────────┬──────────────────────────┐
│ (index) │ Values                   │
├─────────┼──────────────────────────┤
│ 0       │ 10                       │
│ 1       │ 17456272927334475347484n │
│ 2       │ 'Gaurav'                 │
│ 3       │ true                     │
│ 4       │ null                     │
│ 5       │ undefined                │
└─────────┴──────────────────────────┘
```

return typeof each data type -

```
console.log(typeof number1);           number
console.log(typeof number2);           bigint
console.log(typeof name);              string
console.log(typeof isLoggedIn);        boolean
console.log(typeof value);             object
console.log(typeof accountState);      undefined
```

Primitive Data Types -

- String, Number, Boolean, null, undefined, NaN, Symbol, BigInt

Examples of primitive data types -

```
const score = 100                              // number
const PIE = 3.14                               // number
const isLoggedIn = true                        // boolean
const outsideTemp = null                       // null
let userEmail;                                 // undefined
const bigNumber = 1234567890986433335456n      // bigInt
const id = Symbol('123')                       // symbol

const anotherId = Symbol('123')
console.log(id == anotherId);                  // prints false
```

Referenced Types (Non primitive) -

- Array, Objects, Functions

Array -

```
const heros = ["Shaktiman", "Naagraj", "Doga"]
```

Object -

```
let myObj = {
    name: "Gaurav",
    age: 27
}
```

function -

```
const myFunction = function(){
    console.log("Hello Gaurav!");
```

```
}

console.log(typeof heros); // returns object
console.log(typeof myObj); // returns object
console.log(typeof myFunction); // returns function (function object)
```

Memory Types -

- Stack - This memory used in all primitive type data types
- Heap - This memory used in all non-primitive data types
- In stack, we are getting copy of a variable
- In heap the changes made in original values

stack example -

```
let ytChannel = "ChaiAurCode"
let anotherytChannel = ytChannel
anotherytChannel = "TechnicalGuruji"

console.log(ytChannel);          // prints ChaiAurCode
console.log(anotherytChannel);   // TechnicalGuruji
```

heap example -

```
let userOne = {
    email: "gm@yahoo.com",
    upi: "user@axl"
}

let userTwo = userOne

userTwo.email = "gm@google.com"

console.log(userOne.email);      // prints gm@google.com
console.log(userTwo.email);      // prints gm@google.com
```

Conversion Operations

```
let score = 100
console.log(typeof score);       // return number
console.log(typeof(score));      // return number
```

Into number conversion -

```
let myScore = "101"
let valueInMyscore = Number(myScore)
console.log(typeof valueInMyscore);    // return number

let myScore1 = "100abc"
let valueInMyscore1 = Number(myScore1)
console.log(valueInMyscore1);          // return NaN (not a number)

let myScore2 = null
let valueInMyscore2 = Number(myScore2)
console.log(valueInMyscore2);          // returns 0

let myScore3 = undefined
let valueInMyscore3 = Number(myScore3)
console.log(valueInMyscore3);          // returns NaN

let myScore4 = true
let valueInMyscore4 = Number(myScore4)
console.log(valueInMyscore4);          // returns 1

let myScore5 = ""
let valueInMyscore5 = Number(myScore5)
console.log(valueInMyscore5);          // returns 0
```

Values return after converting into Number -

- "101" -> 101
- "100abc" -> NaN
- null -> 0
- undefined -> NaN
- true -> 1, false -> 0
- "" -> 0

Into boolean conversion -

```
let myBoolean = 102
let valueInMyBoolean = Boolean(myBoolean)
console.log(valueInMyBoolean);                // returns true

let myBoolean1 = "Gaurav"
let valueInMyBoolean1 = Boolean(myBoolean1)
console.log(valueInMyBoolean1);               // returns true

let myBoolean2 = null
let valueInMyBoolean2 = Boolean(myBoolean2)
console.log(valueInMyBoolean2);               // returns false

let myBoolean3 = undefined
let valueInMyBoolean3 = Boolean(myBoolean3)
console.log(valueInMyBoolean3);               // returns false

let myBoolean4 = ""
let valueInMyBoolean4 = Boolean(myBoolean4)
console.log(valueInMyBoolean4);               // returns false

let myBoolean5 = 1
let valueInMyBoolean5 = Boolean(myBoolean5)
console.log(valueInMyBoolean5);               // returns true
```

Values returning after conversion into Boolean -

- 101 -> true
- "Gaurav" -> true
- null -> false
- undefined -> false
- "" -> false
- 1 -> true, 0 -> false

Into String conversion -

```
let myString = 103
let valueInMyString = String(myString)
console.log(valueInMyString); // return 103
console.log(typeof(valueInMyString));         // return type is string
```

```
let myString1 = null
let valueInMyString1 = String(myString1)
console.log(valueInMyString1);                // return null

let myString2 = undefined
let valueInMyString2 = String(myString2)
console.log(valueInMyString2);                // return undefined

let myString3 = NaN
let valueInMyString3 = String(myString3)
console.log(valueInMyString3);                // return NaN

let myString4 = true
let valueInMyString4 = String(myString4)
console.log(valueInMyString4);                // return true

let myString5 = " "
let valueInMyString5 = String(myString5)
console.log(valueInMyString5);                // return whitespace
```

Values returning after conversion into string -

- 103 -> 103, return type -> string
- null -> null
- undefined -> undefined
- NaN -> NaN
- true -> true, false -> false
- "" -> whitespace

Operations -

```
console.log("1" + 2);     // prints 12 (string + number = string)
console.log(1 + "2");     // prints 12 (number + string = string)
console.log("1" + 2 + 2); // prints 122 (string + number + number = string)
console.log(1 + 2 + "2"); // prints 32
```

Comparisons

```
console.log(2 = 1);       Error
console.log(2 == 1);      // prints false
console.log(2 > 1);       // prints true
console.log(2 < 1);       // prints false
console.log(2 >= 1);      // prints true
console.log(2 <= 1);      // prints false
console.log(2 != 1);      // prints true

console.log("2" > 1);     // prints true
console.log("02" > 1);    // prints true
```

- The reason is that an equality check == and comparisons > < >= <= work differently.
- Comparison's convert null to a number, treating it as 0.
- That is why (3) null >= 0 is true and (1) null > 0 is false.

```
console.log(null > 0); // prints false
console.log(null == 0); // prints false
console.log(null >= 0); // prints true

console.log(undefined > 0); // false
console.log(undefined == 0); // false
console.log(undefined >= 0); // false
```

- Strict check ===

```
console.log("2" === 2); // false
```

Strings

String Interpolation -> `` (backticks') -

```
let name = "Gaurav"
let repoCount = 10
console.log(`Hi ${name}, your repo count is ${repoCount}.`);

Output : Hi Gaurav, your repo count is 10.
```

Another way of string declaration -

```
let myName = new String("Sumit")
console.log("My name: ", myName);                    // prints [String: 'Sumit']
console.log("First character: ", myName[0]);         // prints first character S
console.log(myName.__proto__);                              // prints empty string
console.log("Length:", myName.length);                      // prints Length: 5
console.log("Upper case: ", myName.toUpperCase());          // prints SUMIT
console.log("Char at index 2: ", myName.charAt([2]));       // prints m
console.log("Index of m: ", myName.indexOf('m'));           // prints index 2

const gameName = "CandyCrush"
const newString = gameName.substring(0, 4)
console.log(newString);                              // prints Cand

const anotherString = gameName.slice(-8, 4)
console.log(anotherString);                          // print nd

const newStringOne = "  Gaurav  "
console.log(newStringOne);                           // prints " Gaurav  "
console.log(newStringOne.trim());                    // prints "Gaurav"

const url = "https://youtube.com/gaurav%20manwatkar"
console.log(url.replace('%20', '-'))
// prints https://youtube.com/gaurav-manwatkar

console.log(url.includes('gaurav'));         // returns true

const anotherGameName = "candy-crush-saga-soda"
console.log(anotherGameName.split('-'));

// prints [ 'candy', 'crush', 'saga', 'soda' ]
```

Numbers -

```
const score = 350
console.log("Score : ", score);        // prints Score :  350

const balance = new Number(100)
console.log("Balance : ", balance);    // prints Balance :  [Number: 100]

console.log("toString : ", balance.toString());     // toString :  100
console.log("typeof balance : ", typeof balance);   // typeof balance :  object

console.log("length of balance : ", balance.toString().length);
// length of balance :  3

console.log("toFixed : ", balance.toFixed(2));      // toFixed :  100.00

const otherNumber = 23.1997
console.log("toPricision 3 : ", otherNumber.toPrecision(3));
// toPricision 3 :  23.2

console.log("toPricision 4 : ", otherNumber.toPrecision(4));
// toPricision 4 :  23.20

const hundreds = 1000000
console.log("toLocaleString : ", hundreds.toLocaleString('en-IN'));
// toLocaleString :  10,00,000
```

Maths -

```
console.log("returns : ", Math);              // returns :  Object [Math] {}
console.log("abs value : ", Math.abs(-4));          // abs value :  4
console.log("round value : ", Math.round(4.6));     // round value :  5
console.log("ceil value: ", Math.ceil(4.2));        // ceil value:  5
console.log("floor value : ", Math.floor(4.7));     // floor value :  4
console.log("min value: ", Math.min(4, 5, 3, 7));   // min value:  3
console.log("max value : ", Math.max(4, 8, 12, 6)); // max value :  12
```

- in random() functions values will be lies in between 0 and 1

```
console.log("random -> toPricision : ", Math.random().toPrecision(4));
// random -> toPricision :  0.8005

console.log("random * 10 + 1 : ", (Math.random()*10) + 1);
// random * 10 + 1 :  3.275674992444301
```

```
const min = 10
const max = 20

console.log("value : ", Math.floor(Math.random() * (max - min + 1)) + min);
// value :  14
```

Dates in javascript

```
let myDate = new Date()
console.log("Return type of Date : ", typeof myDate); // return object

console.log("Date to String : ", myDate.toString());
// Sat Apr 06 2024 17:50:51 GMT+0000 (Coordinated Universal Time)

console.log("Date to Date String : ", myDate.toDateString()); // Sat Apr 06 2024
console.log("Date to Locale String : ", myDate.toLocaleString());
// 4/6/2024, 5:50:51 PM
```

- in javascript months starts from 0

```
let myCreatedDate = new Date(2024, 3, 6)
console.log("toDateString : ", myCreatedDate.toDateString());
// toDateString :  Sat Apr 06 2024

let myCreatedDate1 = new Date(2024, 3, 6, 4, 8, 11)
console.log("toLocaleString : ", myCreatedDate1.toLocaleString());
// toLocaleString :  4/6/2024, 4:08:11 AM

let myCreatedDate2 = new Date("2024-04-06")
console.log("toLocaleString : ", myCreatedDate1.toLocaleString());
// toLocaleString :  4/6/2024, 4:08:11 AM

let myTimeStamp = Date.now()
console.log("timestamp in miliseconds : ", myTimeStamp);
// timestamp in miliseconds :  1712768685975

console.log("Get time : ", myCreatedDate2.getTime());   // Get time :  1712361600000
console.log("Convert to seconds : ", Math.floor(Date.now()/1000));
// Convert to seconds :  1712768685

let newDate = new Date()
console.log("Get month : ", newDate.getMonth());    // Get month :  3
console.log("Get day : ", newDate.getDay());    // Get day :  3

let weekDay = newDate.toLocaleString('default',{
    weekday: "long"
})

console.log("weekday : ", weekDay);     // weekday :  Wednesday
```

Arrays

- Javascript arrays are resizable, mixed of data types, you can add multiple element values
- zero based indexing
- A shallow copy of an object is a copy whose properties share the same reference point (heap)
- A deep copy of an object is a copy whose properties do not share the same reference point (stack)

number array -

```
const myArr = [1, 2, 3, 4, 5]       // 1,2,3,4,5 are called array elements
console.log("0th index: ", myArr[0])    // 0th index:  1
```

string array -

```
const myheros = ["Shaktiman", "Naagraj"]

const myArr1 = new Array(5, 4, 3, 2, 1)
console.log("0th index: ", myArr1[0])   // 0th index:  5
```

Array Methods -

- push() method

```
myArr.push(6)       // add 6 at the end of array
myArr.push(7)       // adds 7 at the end of array
myArr.pop()         // removes last element of an array
console.log("myArr : ", myArr);     // myArr:  [ 1, 2, 3, 4, 5, 6 ]
```

- unshift() method

```
myArr.unshift(0)
console.log(myArr); // adds 0 at the beginning of the array
```

- shift() method

```
myArr.shift()
console.log(myArr); // removes first index of an array
```

- includes() and indexOf()

```
console.log("Is value exist: ", myArr.includes(9));     // Is value exist:  false
console.log("Index of 5: ", myArr.indexOf(5));      // Index of 5:  4
```

- join() method

```
const newArr = myArr.join()
console.log("myArr: ", myArr);  // myArr:  [ 1, 2, 3, 4, 5, 6 ]
console.log("newArr: ", newArr);    // newArr:  1,2,3,4,5,6
console.log("type of myArr: ", typeof myArr);   // type of myArr:  object
console.log("type of newArr: ", typeof newArr); // type of newArr:  string
```

- slice(), splice() methods

```
console.log("A: ", myArr);  // A:  [ 1, 2, 3, 4, 5, 6 ]

const myn1 = myArr.slice(1, 3)

// only prints 1st and 2nd index elements, ignores from end inputed element

console.log("slice result: ", myn1); // slice result:  [ 2, 3 ]

console.log("B: ", myArr);  // B:  [ 1, 2, 3, 4, 5, 6 ]

const myn2 = myArr.splice(1, 3)

// moves the values from original array, moves 1st, 2nd and 3rd elements

console.log("splice result: ", myn2);   // splice result:  [ 2, 3, 4 ]

// splice manipulates the array

console.log("C: ", myArr);  // C:  [ 1, 5, 6 ]
```

More on arrays

```
const marvel_heros = ["Thor", "Ironman", "Spiderman"]
const dc_heros = ["superman", "flash", "batman"]

// here, dc_heros not merged with marvel_heros, its adding entire dc_heros at the
end of marvel_heros (arrays in array)

marvel_heros.push(dc_heros)     // push array dc_heros
console.log("Marvel heros: ", marvel_heros);
// Marvel heros:  [ 'Thor', 'Ironman', 'Spiderman', [ 'superman', 'flash',
'batman' ] ]

console.log("length of marvel_heros: ", marvel_heros.length);
// length of marvel_heros:  4

const marvel_heros1 = ["Thor", "Ironman", "Spiderman"]
const dc_heros1 = ["superman", "flash", "batman"]

const allHeros = marvel_heros1.concat(dc_heros1)    // concat() method
console.log("All heros: ", allHeros);
// All heros:  [ 'Thor', 'Ironman', 'Spiderman', 'superman', 'flash', 'batman' ]
```

- ... spread operator

```
const allNewHeros = [...marvel_heros1, ...dc_heros1]
console.log("All new heros: ", allNewHeros);
// All new heros:  [ 'Thor', 'Ironman', 'Spiderman', 'superman', 'flash',
'batman']
```

- flat() method

```
const anotherArr = [1, 2, 3, [4, 5, 6], 7, [8, 9, [0, 1, 2]]]
const realAnotherArr = anotherArr.flat(Infinity)
console.log("Real another array: ", realAnotherArr);
```

- isArray() and from()

```
console.log("isArray:", Array.isArray("Gaurav")); // isArray: false

console.log("from:", Array.from("Gaurav"));
// from: [ 'G', 'a', 'u', 'r', 'a', 'v' ]

console.log("from:", Array.from({name: "Gaurav"})); // from: []
```

- Array.of()

```
let score1 = 100
let score2 = 200
let score3 = 300

console.log("Array of: ", Array.of(score1, score2, score3));
// Array of:  [ 100, 200, 300 ]
```

Objects

- Singleton - made from constructor (Object.create)

Object literals -

```
const mySymbol = Symbol("key1")

const jsUser ={
    name: "Gaurav",
    "full name": "Gaurav Manwatkar",
    [mySymbol]: "mykey1",
    age: 20,
    location: "Nagpur",
    email: "gm@gmail.com",
    isLoggedIn: false,
    lastLoginDay: ["Monday", "Saturday"]
}

console.log("Email:", jsUser.email);    // Email: gm@gmail.com
console.log("Email:", jsUser["email"]); // Email: gm@gmail.com
console.log("full name:", jsUser["full name"]); // full name: Gaurav Manwatkar
console.log("mySymbol:", jsUser[mySymbol]); // mySymbol: mykey1

jsUser.email = "gm@yahoo.com"
console.log("updated email:", jsUser.email);    // updated email: gm@yahoo.com

jsUser.greeting = function(){
    console.log("Hello js users....");
}
console.log(jsUser.greeting());     // Hello js users....

jsUser.greetingTwo = function(){
    console.log(`Hello js user, ${this.name}`);
}
console.log(jsUser.greetingTwo());      // Hello js user, Gaurav
```

more on objects -

- creating tinderUser object

```
const tinderUser = new Object()
tinderUser.id = "123abc"
tinderUser.name ="SAM"
tinderUser.isLoggedIn = false

console.log("tinderUser object : ", tinderUser);
// tinderUser object :  { id: '123abc', name: 'SAM', isLoggedIn: false }
```

- userfullname object inside fullname object (objects in object)

```
const regularUser = {
    email: "gm@gmail.com",
    fullname: {
        userfullname: {
            firstname: "Gaurav",
            lastname: "Manwatkar"
        }
    }
}

console.log(regularUser.fullname);
// { userfullname: { firstname: 'Gaurav', lastname: 'Manwatkar' } }

console.log(regularUser.fullname.userfullname);
// { firstname: 'Gaurav', lastname: 'Manwatkar' }

console.log(regularUser.fullname.userfullname.firstname);   // Gaurav
```

- assigning objects to objects

```
const obj1 = {1: "a", 2: "b"}
const obj2 = {3: "a", 4: "b"}
const obj3 = Object.assign({}, obj1, obj2)
const obj4 = {...obj1, ...obj2}

console.log("obj3 : ", obj3);   // obj3 :  { '1': 'a', '2': 'b', '3': 'a', '4': 'b' }
console.log("obj4 : ", obj4);   // // obj4 :  { '1': 'a', '2': 'b', '3': 'a', '4': 'b' }



console.log("tinderUser : ", tinderUser);
```

```
// tinderUser :  { id: '123abc', name: 'SAM', isLoggedIn: false }

console.log("keys : ", Object.keys(tinderUser));    // keys :  [ 'id', 'name', 'isLoggedIn' ]
console.log("values : ", Object.values(tinderUser));    // values :  [ '123abc', 'SAM', false ]

console.log("entries : ", Object.entries(tinderUser));
// entries :  [ [ 'id', '123abc' ], [ 'name', 'SAM' ], [ 'isLoggedIn', false ] ]

console.log("hasOwnProperty : ", tinderUser.hasOwnProperty("isLoggedIn"));
// hasOwnProperty :  true
```

De-structuring

```
const course = {
    coursename: "js in hindi",
    price: "999",
    courseInstructor: "Gaurav"
}

console.log("course instructor : ", course.courseInstructor) // course instructor :  Gaurav

const {courseInstructor : instructor} = course // destructuring of object
console.log("Course Instructor : ", instructor); // Course Instructor : Gaurav
```

Functions

function example -

```
function sayMyName(){
    console.log("G");
    console.log("A");
    console.log("U");
    console.log("R");
    console.log("A");
    console.log("V");
}

sayMyName()
```

add two numbers using functions -

- number1, number2 are called parameters

```
function addTwoNumbers(number1, number2){
    console.log("Addition:", number1 + number2);    // Addition: 10
}

addTwoNumbers(5, 5) // 5, 5 are called arguments
```

another way of writing function -

```
function addTwoNumbers1(number1, number2){
   const result = number1 + number2
   return result
}
const result = addTwoNumbers1(6, 6)
console.log("Result: ", result);        // Result:  12
```

passing string as parameter -

```
function loginUserMessage(username){
    return `${username}, just logged in....`
}
console.log(loginUserMessage("gaurav"));    // gaurav, just logged in....
```

- when value is not passed to function it returns undefined

```
console.log(loginUserMessage());    // undefined, just logged in....
```

... rest operator

```
function calculateCartPrice(...num1){       // ... rest operator
    return num1
}
console.log("cart values : ", calculateCartPrice(200, 300, 500));
// cart values :  [ 200, 300, 500 ]
```

passing object to the function

```
const user = {
    username: "gaurav",
    price: 199
}

function handleObject(anyObject){
    console.log(`username is ${anyObject.username} and the price is
${anyObject.price}`);
// username is gaurav and the price is 199
}

handleObject(user) // passing object "user" to function "handleObject"
```

another way of passing object to the function -

```
handleObject({
    username: "Faizal",
    price: 399
})      // username is Faizal and the price is 399
```

passing array to the function -

```
const myNewArr = [100, 200, 300, 400, 500]

function returnValue(getArr){
    return getArr[2]
}
console.log("return array element: ", returnValue(myNewArr));
// return array element:  300
```

another way of passing array to the function -

```
console.log("new element: ", returnValue([100, 200, 300, 400, 500]));   // new element:  300
```

scope in javascript

```
let a = 40

if(true){
    let a = 10
    const b = 20
    //var c = 30
    console.log("inner a: ", a); // local scope - inner a:  10
}

// console.log(a); // error cause declared local scope
// console.log(b); // declared in local scope
// console.log(c); // prints 30 because var can be accessed anywhere

console.log("outer a: ", a); // global scope - outer a:  40
```

- note : in node environment scope acts differently, also same for windows environment (in ide level)

```
function one(){
    const username = "gaurav"

    function two(){
        const website = "youtube"
        console.log("username:", username);     // username: gaurav
    }
    // console.log(website);
    two()
}
one()
```

if scope -

```
if(true){
    const username = "gaurav"
    if(username === "gaurav"){
        const website = " youtube"
        console.log(username + website);    // gaurav youtube
    }
    // console.log(website);
}
// console.log(username);
```

Interesting -

```
function addOne(num){
    return num + 1
}

addOne(5)

// function cannot access before initialization
// addTwo(5) -> if we call the function here
const addTwo = function(num){       // addTwo is also called expression
    return num + 2
}

addTwo(5)
```

Immediate Invoked Function Expressions (IIFE)

- (function definition) (execution)

- In IIFI we have to wrap a function in () and have to add () at the end of the line as shown below

```
(function db1(){
    // db1 is a named IIFE, as function name is given
    console.log(`DB1 connected...`);    // DB1 connected...
})(); // to end the execution we have to give ; at the end

// to remove the global declaration polution we have to use IIFE

(() => {
    console.log(`DB2 connected...`);    // DB2 connected...
})();

// parameter to arrow function IIFE

((name) => {
    console.log(`DB3 connected to ${name}...`); // DB3 connected to gaurav...
})("gaurav")
```

Javascript Execution Context

```
{
    // Execution Context Types
    Global Execution Context
    Function Execution Context
    Eval Execution Context (property of gloabl object)
}
```

Execution of Javascript Execution Context -

Executes in 2 phases:

1 -> Memory Creation Phase (allocates space for variables, functions, etc)
2 -> Execution Phase

Example of Memory Creation Phase -

```
    let val1 = 10
    let val2 = 5
    function addNum(num1, num2){
        let total = num1 + num2
        return total
    }
    let result1 = addNum(val1, val2)
    let result2 = addNum(5, 10)

    // Execution of Memory Creation Phase

    {
        1. Global Execution allocates to 'this'

        2. Memory Phase (collects all the variables and allocates the memory)
            First cycle{
            - val1 -> undefined
            - val2 -> undefined
            - addNum -> function definition
            - result1 -> undefined
            - result2 -> undefined}




        3. Execution Phase
```

```
        {
            val1 -> 10
            val2 -> 5
            addNum -> creates own Execution Context
                                                    {
                                                        New Variable Environment
                                                                    +
                                                            Execution Thread
                                                        (this execution context will be deleted
once executed)
                                                    }

                                                    // phases for addNum
                                                    1. Memory Phase
                                                        {
                                                            val1 -> undefined
                                                            val2 -> undefined
                                                            total -> undefined
                                                        }

                                                    2. Execution Phase
                                                        {
                                                            num1 -> 10
                                                            num2 -> 5
                                                            total -> 15
                                                            // total returns in GEC
                                                        }
            result1 = 15
            addNum -> creates own Execution Context
                                                    {
                                                        New Variable Environment
                                                                    +
                                                            Execution Thread
                                                        (this execution context will be deleted
once executed)
                                                    }

                                                    // phases for addNum
                                                    1. Memory Phase
                                                        {
                                                            val1 -> undefined
                                                            val2 -> undefined
                                                            total -> undefined
                                                        }
```

```
2. Execution Phase
    {
        num1 -> 5
        num2 -> 10
        total -> 15
        // total returns in GEC
    }
```

```
     }
```

Call Stack Example :

```
// in call stack LIFO style follows
    {
        function one(){
            console.log("One)
            two()
        }

        function two(){
            console.log("Two)
            three()
        }

        function three(){
            console.log("Three)
        }

        one()
        two()
        three()
    }

}
```

if statement in javascript

comparison operators -

- <, >, <=, >=, ==, !=, === (also checks type), !== (-ve sign check)

```
const isUserLoggedIn = true
const temperature = 50

if(temperature === 55){
    console.log("less than 50");
} else{
    console.log("temperature is greater than 55");
      // temperature is greater than 55
}

const score = 200
const power = "fight"

if(score > 100){
    const power = "fly"
    console.log(`user power: ${power}`);    // user power: fly
}

console.log(`user power: ${power}`);    // user power: fight
```

short hand notation -

```
const balance = 1000
if (balance > 500) console.log("true");     // true

// nesting

if (balance < 500) {
    console.log("less than 500");
} else if (balance < 750) {
    console.log("less than 700");
} else if (balance < 900) {
    console.log("less than 900");
}else{
    console.log("less than 1200");  // less than 1200
}
```

condition check via logical operators -

```
const userLoggedIn = true
const debitCard = true
const loggedInFromGoogle = false
const loggedInFromEmail = true

if (userLoggedIn && debitCard) {
    console.log("allow to buy courses");    // allow to buy courses
}

if (loggedInFromGoogle || loggedInFromEmail) {
    console.log("user logged in");  // user logged in
}
```

switch statement

passing number -

```
const month = 3
switch (month) {
    case 1:
        console.log("january");
        break;
    case 2:
        console.log("feb");
        break;
    case 3:
        console.log("march");   // march
        break;
    case 4:
        console.log("april");
        break;

    default:
        console.log("default case match");
        break;
}
```

passing string -

```
const myMonth = "march"
switch (myMonth) {
    case "january":
        console.log("january");
        break;
    case "feb":
        console.log("feb");
        break;
    case "march":
        console.log("march");   // march
        break;
    case "april":
        console.log("april");
        break;

    default:
        console.log("default case match");
        break;
}
```

truthy and falsy values

```
const userEmail = "gm@gmail.com"

if (userEmail) {
    console.log("got user email");  // got user email
} else{
    console.log("dont have user email");
}
```

- we assumes that the given string is a true value thats the truthy value.
- falsy values -> false, 0, -0, bigInt 0n, "", null, undefined, NaN
- truthy values -> "0", 'false', " ", [], {}, function(){}

empty arrays checks -

```
const emptyArray = []

if (emptyArray.length === 0) {
    console.log("array is empty");  // array is empty
}
```

empty object checks -

```
const myObject = {}

if (Object.keys(myObject).length === 0) {
    console.log("object is empty");     // object is empty
}
```

- false == 0 -> true, false == '' -> true, 0 == '' -> true

Nullish Colescing Operator (??): null undefined -

```
let val1 = 5 ?? 10
val1 = null ?? 10
val1 = undefined ?? 15
val1 = null ?? 10 ?? 25

console.log(val1);  // 10
```

Terniary operator (?) -

```
// condition ? true : false

const iceTeaPrice = 100

iceTeaPrice <= 80 ? console.log("less that 80") : console.log("more than 80");
// more than 80
```

for loop on number array -

```
for (let i = 0; i <= 10; i++) {
    const element = i;
    if (element == 5) {
        console.log("5 is the best number");
    }
    console.log(element);
}

for (let i = 1; i <= 10; i++) {
    console.log(`table of : ${i}`);
    for (let j = 1; j <= 10; j++) {
        //console.log(`inner loop value ${j} and inner loop ${i}`);
        console.log(i + ' * ' + j + ' = ' + i*j);
    }
}
```

for loop on string arrays -

```
let myArray = ["flash", "batman", "superman"]
    console.log("myArray length: ", myArray.length);

for (let i = 0; i < myArray.length; i++) {
    const element = myArray[i];
    console.log("myArray value:", element);
}
```

break and continue -

```
for (let i = 1; i <= 20; i++) {
    if (i == 5) {
        console.log("5 detected!");
        break
```

```
        }
        console.log(`value of i is ${i}`);
    }

    for (let i = 1; i <= 20; i++) {
        if (i == 5) {
            console.log("5 detected!");
            continue // 1 times condition skipped
        }
        console.log(`value of i is ${i}`);
    }
```

while loop -

```
    let index = 0
    while (index <= 10) {
        console.log(`value of index is ${index}`);
        index += 2
    }

    let myArray = ["flash", "batman", "superman"]

    let arr = 0
    while (arr < myArray.length) {
        console.log(`value is ${myArray[arr]}`);
        arr++
    }
```

do while loop -

```
let score = 1

do {
    console.log(`score is ${score}`);
    score++
} while (score <= 10);
```

for loop on array and map() -

```
// ["", "", ""]
// [{}, {}, {}]

                    const arr = [1, 2, 3, 4, 5]

                    for (const ar of arr) {
                        console.log(ar);    // prints 1 to 5
                    }

                    const greetings = "hello world"

                    for (const grret of greetings) {
                        console.log(`each char is ${grret}`);   // print each character on new
                    line
                    }
```

Maps -

```
                    const map = new Map()
                    map.set('IN', 'India')
                    map.set('USA', 'United States of America')
                    map.set('FR', "France")

                    console.log(map);   // prints map

                    for (const [key, value] of map) {
                        console.log(key, '->', value);  // print in key : value form
                    }
```

loop on object -

```
                    const myObj = {
                        'game1': 'NFS',
                        'game2': 'spiderman',
                        'game3': 'contra'
                    }
                    /*  TypeError: myObj is not iterable

                        for (const [key, value] of myObj) {
                            console.log(key, '->', value);
                        }
                    */
```

accessing values via key -

```
const myObj = {
    js: 'javascript',
    cpp: 'C++',
    rb: 'ruby',
    swift: 'swift by apple'
}
```

accessing values in objects via key -

```
for (const key in myObj) {
   console.log(`${key} shortcut is for ${myObj[key]}`);
}
```

accessing values in array via keys -

```
const programming = ['js', 'ruby', 'py', 'java', 'cpp']

for (const key in programming) {
console.log(programming[key]);
}
```

accessing values in maps via keys -

```
const map = new Map()
map.set('IN', 'India')
map.set('USA', 'United States of America')
map.set('FR', "France")

for (const key in map) {
    console.log(key);
}
```

for each loop

```
const coading = ["js", "ruby", "java", "python", "cpp"]

coading.forEach(function (item) {
    console.log(item);
} ) // call back function dont have any name

coading.forEach( (item) => {
    console.log(item);
})

function printMe(item){
    console.log(item);
}

coading.forEach(printMe)

coading.forEach( (item, index, arr) => {
    console.log(item, index, arr);
})

const myCoading = [
    {
        languageName: "javascript",
        languageFileName: "js"
    },
    {
        languageName: "java",
        languageFileName: "java"
    },
    {
        languageName: "python",
        languageFileName: "py"
    }
]

myCoading.forEach((item) => {
    console.log(item.languageName);
})
```

using forEach() -

```
const coading = ["js", "ruby", "java", "python", "cpp"]

const values = coading.forEach( (item) => {
    console.log(item);
    return item
})

console.log(values);
// for each does not return anything, its only prints undefined
```

numbers – filter -

```
const myNums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

const newNums = myNums.filter( (num) => num > 5) // filter returns the
values hence need to store in variable to print them
console.log(newNums);

const newNums1 = myNums.filter( (num) => {
    return num > 5
})
console.log(newNums);
```

using for each -

```
const newNums2 = []
myNums.forEach( (nums) => {
    if (nums > 5) {
        newNums2.push(nums)
    }
})

console.log(newNums2);
```

books array example -

```
const books = [
    { title: 'Book One', genre: 'Fiction', publish: 1981, edition: 2004 },
    { title: 'Book Two', genre: 'Non-Fiction', publish: 1992, edition: 2008 },
    { title: 'Book Three', genre: 'History', publish: 1999, edition: 2007 },
    { title: 'Book Four', genre: 'Non-Fiction', publish: 1989, edition: 2010 },
    { title: 'Book Five', genre: 'Science', publish: 2009, edition: 2014 },
    { title: 'Book Six', genre: 'Fiction', publish: 1987, edition: 2010 },
    { title: 'Book Seven', genre: 'History', publish: 1986, edition: 1996 },
    { title: 'Book Eight', genre: 'Science', publish: 2011, edition: 2016 },
    { title: 'Book Nine', genre: 'Non-Fiction', publish: 1981, edition: 1989 },
  ];

  const userBooks = books.filter( (bk) => bk.genre === "History")
  let userBooks1 = books.filter( (bk) => bk.publish >= 2000)
  let userBooks2 = books.filter( (bk) => bk.publish >= 1995 && bk.genre === "History")

  console.log(userBooks);
  console.log(userBooks1);
  console.log(userBooks2);
```

chaining in javascript -

```
const myNums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

const newNums = myNums.map((num) => num + 10) // map automatically return values
console.log(newNums);

// chaining

const newNums1 = myNums
                    .map((nums) => nums * 10)
                    .map( (nums) => nums + 1)
                    .filter((nums) => nums >= 40)
console.log(newNums1);
```

reduce function -

```
const myNums = [1, 2, 3, 4, 5]

const myTotal = myNums.reduce( function (acc, curval) {
    console.log(`acc: ${acc} and curval: ${curval}`);
    return acc + curval
}, 0)

console.log(myTotal);   // 15

const myTotal1 = myNums.reduce( (acc, curval) => acc+curval, 0)
console.log(myTotal1);  // 15

const shoppingCart = [
    {
        itemname: "js course",
        price: 999
    },
    {
        itemname: "mobile dev course",
        price: 1999
    },
    {
        itemname: "js course",
        price: 9999
    }
]

const priceToPay = shoppingCart.reduce( (acc, item) => acc + item.price, 0)
console.log(priceToPay);    // 12997
```