**Array :-**

1. Array store the collection which can accessing using the indexing
2. It contain heterogeneous data
3. Converting the array which storing the value of number type we have to convert them into the string using toString() method
4. Arrays have some properties and the method

   -> array.length // returns the length of the array

   -> array.sort() // sort is the method inside the array which is sorting in case the heterogeneous data

   -> array.push("New Element") // adding the new element to array

   -> array.fill(0,2,4) // fill 0 value from index [2,4)

   -> array.filter(array(word)=>word.length>6); // filteraccording to the condition

   -> array. findIndex(3); // giving the index of the value present and -f not -1

   -> array.findLast(140); // from last index we have to find the particular element

   -> array.flat(infinity): // flat the array

   -> array.join('-'); // join the all the elements of the array

   -> array.slice(2,5); // slicing according to the index(2,5]
5. Arrays uses numbered indexed not the named index

   const array = new Array(10); // helping in the creating the array of the size 10 having undefined value

   const array=["90",0]; // directly assigned so that not to mention the size of the array

   const array = new Array("Hi","Hello");

   Important Note : holes -> adding the elements at the high index created holes in the array

**Operators :-**

1. Assignment operator -> helping in assign the value
2. Addition operator -> adding the value
3. Multiplication operator -> multiply the two values
4. Comparison operator ->

   == / using the check the equality of the value

   === / using and check the equality as well as the data type

   != / not equal to

   !== / not equal value or not equal datatype

   > / greater than

   < / smaller than

   >= / greater than or equal to

   <= / smaller than or equal to

   ?: / ternary operator

**Execution Context:**

JavaScript execution context is a crucial part of understanding how JavaScript works behind the scenes.

It determines the environment in which code is executed and what variables and functions are available to use.

The creation phase includes creating the global and function execution contexts, creating the scope chain, and allocating memories for the variables and functions.

During the execution phase, the JavaScript engine executes the code line by line. This includes evaluating and executing statements.

Execution context is the concept for describing the internal working of a code. In JavaScript, the environment that enables the JavaScript code to get executed is what we call JavaScript Execution Context.

It is the execution context that decides which code section has access to the functions, variables, and objects used in the code. During the execution context,

the specific code gets parsed line by line then the variables and functions are stored in the memory.

An execution context is similar to a container that stores variables, and the code gets evaluated and then executed. Thus, it is the execution context that provides an environment for the specific code to get executed.

There are three types of Execution Context:

1. Global Execution Context

2.Functional Execution Context

3.Eval Execution Context

**Callback Queue:**

Asynchronous operations, such as I/O operations or timers, are handled by the browser or Node.js runtime. When these operations are complete, corresponding functions (callbacks) are placed in the callback queue.

**Event Loop:**

The event loop continuously checks the call stack and the callback queue. If the call stack is empty, it takes the first function from the callback queue and pushes it onto the call stack for execution.

**Execution:**

The function on top of the call stack is executed. If this function contains asynchronous code, it might initiate further asynchronous operations.

**Callback Execution:**

When an asynchronous operation is complete, its callback is placed in the callback queue.

**Repeat:**

The event loop continues this process, ensuring that the call stack is always empty before taking the next function from the callback queue.