JavaScript is a programming language used to make web pages interactive. It's essential for creating dynamic content and real-time updates, enchancing the user experience on websites.
Execution Context :
Execution context in JavaScript is the environment where the code is executed. It consists of :
Two types of memories :

- Call Stack : Call Stack manages function calls, It follows a "Last In First Out" (LIFO) structure. When a function is called, it's added to the stack, when completed, It's removed.
- Memory Heap: Used for storing objects and variables. It is an unstructured region of memory where memory allocation happens dynamically.

Dynamic Allocation :

Dynamic Allocation means allocating memory at runtime as needed, rather than at complie time. In JS, this allows variables , objects, and data structures to be created and resized as the program runs, making memory use more flexible and efficient.

Data Types :

In JavaScript there are two types of data Types

1. Primitive data Types
    - Number , String, Boolean
    - Undefined : A variable is declared but not assigned a value
    - Null : A variable explicitly assigned to indicate no value.
    - Symbol : A unique identifier.
    - BigInt : For large integers.
2. Non-Primitive Types
    - Object
    - Arrays
    - Function

typeof :

Used to check the type of a variable or value.

```
var num = 42;
console.log(typeof num); // Output: "number"

var str = "hello";
```

```
console.log(typeof str); // Output: "string"
```

Key Words :

A keyword is a reserved word in a programming language that has a special meaning and is used to perform specific operations. Keywords are part of the syntax and cannot be used for variable names. There are three types of keywords

1. `var` : It is globally scoped. Can be re-assigned and re-declared within its scope.
2. `let` : It is Block-scoped. Can be re-assigned but not re-declared within the same scope.
3. `const` : It's value is constant and cannot be changed, but if it holds an object or array you can modify its contents but you cannot re-assign the variable itself.

Operators :

JavaScript has several operators, including:

- Arithmetic operators: `+`, `-`, `*`, `/`, `%`, etc.
- Comparison operators: `==`, `!=`, `===`, `!==`, `>`, `<`, etc.
- Logical operators: `&&`, `||`, `!`, etc.
- Assignment operators: `=`, `+=`, `-=`, `*=`, `/=`, etc.

Scope :

Scope refers to the visibility and accessibility of variables and functions in different parts of your code.

- **Global Scope**: Variables declared outside any function are in the global scope and can be accessed from anywhere in the code.
- **Local Scope**: Variables declared inside a function or block are in the local scope and can only be accessed within that function or block.

```
// Example of Scope
var a = 5;
function foo() {
  var b = 2;
  console.log(a); // Output: 5
  console.log(b); // Output: 2
}
foo();
console.log(a); // Output: 5
```

```
console.log(b); // Error: b is not defined
```

Scope Chain :

The chain of scopes that the JavaScript engine uses to resolve variable names. It starts from the innermost scope and works its way outwards. If a variable isn't found in the current scope, JavaScript looks in the outer (enclosing) scopes until it reaches the global scope.

```
let globalVar = 'Global';
function outerFunction() {
    let outerVar = 'Outer';

    function innerFunction() {
        let innerVar = 'Inner';
        console.log(innerVar);   // Accessible (innermost scope)
        console.log(outerVar);   // Accessible (outer scope)
        console.log(globalVar);  // Accessible (global scope)
    }

    innerFunction();
}
outerFunction();
console.log(innerVar);  // Error: innerVar is not defined
```

Object Prototype :

In JavaScript, every object has a prototype, which is another object that provides properties and methods to the original object.

```
function Person(name) {
  this.name = name;
}

// Add method to prototype so it's shared by all Person instances
Person.prototype.greet = function() {
  console.log(`Hello, my name is ${this.name}`);
};

const alice = new Person('Alice');
const bob = new Person('Bob');
alice.greet(); // Hello, my name is Alice
bob.greet();   // Hello, my name is Bob
```

You use **prototypes** in JavaScript when you want to create objects that share properties or methods without duplicating them for each object instance.

Type of Coercion :

There are two types of Coercion :

- Implicit Coercion : Automatically happens when you use operators.

```
console.log(5 + '10');  // "510" (number 5 is coerced to a string)
console.log('5' * 2);   // 10 (string '5' is coerced to a number)
```

- Explicit Coercion: Manually converting a value from one type to another using functions or operators.

```
console.log(Number('5')); // 5 (string '5' is explicitly converted to number)
console.log(String(10));  // "10" (number 10 is explicitly converted to
string)
```

Difference between Nan, undefined, null :

- NaN : Represents a value that is not a valid number.

```
console.log(0 / 0);         // NaN
console.log(parseInt('abc')); // NaN
```

`Note':` NaN `is not equal to any value, including itself (` NaN === NaN `is` false `).`

- undefined : Indicates that a variable has been declared but not yet assigned a value

```
let x;
console.log(x); // undefined (variable declared but not assigned)

function test() {}
console.log(test()); // undefined (function does not return a value)
```

- null : Represents the intentional absence of any object value. It is used to explicitly indicate that a variable should have no value.

```
let obj = null;
console.log(obj); // null
```

Strict Mode :

Without strict mode, JavaScript **automatically creates global variables** if you assign values to undeclared variables. This can lead to **bugs** and **unintended behavior**, as global variables can be modified anywhere in the code. Strict mode prevents this by throwing errors for undeclared variables.

```
"use strict";
function example() {
  x = 3.14; // Error: x is not defined
}
example();
// if there is no strict mode it will automatically create global variable
```

Hoisting :

Hoisting in JavaScript means that function and variable declarations are moved to the top of their scope (global or function scope) during the compilation phase, before the code is executed.

```
sayHello(); // Output: Hello!
function sayHello() {
  console.log('Hello!');
}
```

Functions :

There are several types of functions in JavaScript

1. `Function Declaration or Named Function` : A function with a name that can be called anywhere in the code because it's hoisted.

```
function greet() {
  console.log('Hello!');
}
greet(); // Output: Hello!
```

2. `Function Expression` : A function stored in a variable. It's not hoisted, so you can only call it after the definition.

```
const greet = function() {
  console.log('Hi!');
};
greet(); // Output: Hi!
```

3. `Arrow Function` : A shorter way to write functions. It doesn't have its own `this` context, and is often used for shorter functions.

```
const greet = () => console.log('Hey!');
greet(); // Output: Hey!
```

4. `Anonymous Function` : A function without a name. Typically used when you pass a function directly as an argument.

```
setTimeout(function() {
  console.log('Hello after 1 second');
}, 1000);
```

5. `Callback Function` : A function that is passed as an argument to another function and executed later.

```
function process(callback) {
  callback();
}
process(function() {
  console.log('Callback executed!');
});
```

6. `Higher-Order-Function(HOF)` : Higher-Order Function is a function that takes another function as an argument or returns a function as a result.

```
function higherOrderFunction() {
  return function() {
    console.log('Returned function executed!');
  };
}
const fn = higherOrderFunction();  // HOF returns a function
fn();  // Output: 'Returned function executed!'
```

Parameters vs. Arguments

`Parameter` : These are placeholders in a function definition. They act as variables that will receive values when the function is called.

```
function greet(name) { // 'name' is a parameter
  console.log('Hello, ' + name);
}
```

`Arguments` : These are the actual values you pass into the function when you call it. Arguments are assigned to the parameters.

```
greet('Alice'); // 'Alice' is the argument
```

'This' keyword :

he `this` keyword refers to the current object in the execution context.

```
// Example of ‘this’ keyword
var person = {
  name: "John",
  age: 30,
  greet: function() {
    console.log("Hello, my name is " + this.name + " and I am " + this.age + "
years old.");
  }
};

person.greet(); // Output: "Hello, my name is John and I am 30 years old."
```

Closures :

A closure is created when a function retains access to its lexical scope, even after the function has finished executing. This means that an inner function retains access to variables from its outer (enclosing) function.

```
function outerFunction() {
  let outerVariable = 'I am outside!';

  function innerFunction() {
    console.log(outerVariable); // innerFunction() has access to outerVariable
```

```
  }

  return innerFunction;
}

const closureFunction = outerFunction();
closureFunction(); // Output: I am outside!
```

Lexical Scoping ;

Lexical scoping means that the scope of a variable is determined by its position in the source code. The variables are accessible within the block of code where they are defined and any nested blocks.

Immediately Invoked Function Expression (IIFE) :

An IIFE (Immediately Invoked Function Expression) is a function that is executed immediately after it is defined, we can't reuse it.

```
// Example of IIFE
(function() {
  console.log("I am an IIFE.");
})();
```

call(), apply() and bind() methods

These methods are used to call a function with a specific context and arguments.

- `call()` : Calls a function with a specific context and arguments e.g. `add.call(this,1,2);`
- `apply()` : Calls a function with a specific context and an array of arguments e.g. `add.apply(this,[1,2,]) ;`
- `bind()` : Creates a new function that is bound to a specific context and arguments, e.g. `var boundAdd = add.bind(this,1,2);`

```
var person = {
  name: "John",
  age: 30
};

function greet(greeting) {
  console.log(greeting + ", my name is " + this.name + " and I am " + this.age
```

```
+ " years old.");
}

greet.call(person, "Hello"); // Output: "Hello, my name is John and I am 30
years old."
greet.apply(person, ["Hello"]); // Output: "Hello, my name is John and I am 30
years old."
var boundGreet = greet.bind(person);
boundGreet("Hello"); // Output: "Hello, my name is John and I am 30 years
old."
```

Currying :

Currying is a technique in functional programming where a function with multiple parameters is transformed into a series of functions, each taking a single parameter. It allows you to create specialized functions by fixing some arguments in advance.

```
function add(a) {
  return function(b) {
    return function(c) {
      return a + b + c;
    };
  };
}
// Use the curried function
const result = add(1)(2)(3);  // Computes 1 + 2 + 3
console.log(result);  // Output: 6
```

It allows you to create specialized functions by fixing some arguments in advance. means like below code

```
function multiply(a) {
  return function(b) {
    return a * b;
  };
}
const double = multiply(2);  // Fixes 'a' to 2
```

```
const result = double(5);     // Uses 'b' as 5
console.log(result);         // Output: 10
```

- `multiply(2)` fixes `2` as the first argument (`a`).
- `double` is now a specialized function that will always multiply by `2`.
- When you call `double(5)`, it multiplies `2` (the fixed value) by `5` (the new input), returning `10`.

```
// Example of Currying
function add(a) {
  return function(b) {
    return a + b;
  }
}

var add5 = add(5);
console.log(add5(2)); // Output: 7
```

String, Math and Date objects & methods:

These objects provide various methods for working with strings, numbers and dates.

- String: `length`, `charAt()`, `indexOf()`, `split()`, etc.
- Math: `PI`, `E`, `pow()`, `sqrt()`, `random()`, etc.
- Date: `getFullYear()`, `getMonth()`, `getDate()`, `getHours()`, etc.

```
// String
var str = "hello";
console.log(str.length); // Output: 5
console.log(str.charAt(0)); // Output: "h"
console.log(str.indexOf("l")); // Output: 2
console.log(str.split("")); // Output: ["h", "e", "l", "l", "o"]

// Math
console.log(Math.PI); // Output: 3.141592653589793
console.log(Math.E); // Output: 2.718281828459045
console.log(Math.pow(2, 3)); // Output: 8
console.log(Math.sqrt(16)); // Output: 4
console.log(Math.random()); // Output: a random number between 0 and 1

// Date
var date = new Date();
console.log(date.getFullYear()); // Output: the current year
```

```
console.log(date.getMonth()); // Output: the current month
console.log(date.getDate()); // Output: the current day of the month
console.log(date.getHours()); // Output: the current hour
```

Array & Object properties and methods

These objects provide various methods for working with arrays and objects.

- Array: `length`, `push()`, `pop()`, `shift()`, `unshift()`, etc.
- Object: `keys()`, `values()`, `entries()`, `hasOwnProperty()`, etc.

```
// Array
var arr = [1, 2, 3, 4, 5];
console.log(arr.length); // Output: 5
arr.push(6);
console.log(arr); // Output: [1, 2, 3, 4, 5, 6]
arr.pop();
console.log(arr); // Output: [1, 2, 3, 4, 5]
arr.shift();
console.log(arr); // Output: [2, 3, 4, 5]
arr.unshift(1);
console.log(arr); // Output: [1, 2, 3, 4, 5]

// Object
var obj = { name: "John", age: 30, city: "New York" };
console.log(Object.keys(obj)); // Output: ["name", "age", "city"]
console.log(Object.values(obj)); // Output: ["John", 30, "New York"]
console.log(Object.entries(obj)); // Output: [["name", "John"], ["age", 30"],
["city", "New York"]]
console.log(obj.hasOwnProperty("name")); // Output: true
console.log(obj.hasOwnProperty("gender")); // Output: false
```

DOM(Document Object Model) :

The **DOM** is a programming interface for web documents. It represents the structure of a webpage and allows JavaScript to interact with and manipulate HTML elements.

DOM Events:

Events in the DOM are actions or occurrences that happen in the browser (like clicking a button, submitting a form, or loading a page). JavaScript can respond to these events by executing code.

Common Events:

- `click` : Triggered when an element is clicked.
- `keydown` : Triggered when a key is pressed.
- `mouseover` : Triggered when the mouse pointer is moved onto an element.

```
document.getElementById('myButton').addEventListener('click', function() {
  alert('Button clicked!');
});
```

DOM properties :

Properties allow you to get or set values of DOM elements. These properties reflect the attributes or state of HTML elements.

Common Properties:

- `innerHTML` : The content inside an element (including HTML).
- `value` : The value of an input field.
- `style` : Allows you to change the CSS styles of an element.
- `src` : The source of an image or script tag.

DOM Methods :

Methods are functions that allow you to manipulate the DOM by selecting, creating, or modifying elements.

Common Methods:

- `getElementById()` : Selects an element by its ID.
- `querySelector()` : Selects the first element that matches a CSS selector.
- `createElement()` : Creates a new HTML element.
- `appendChild()` : Adds a new child element to a parent element.

```
// Properties
var div = document.getElementById("myDiv");
console.log(div.innerHTML); // Output: the inner HTML of the div
console.log(div.outerHTML); // Output: the outer HTML of the div
console.log(div.style.color); // Output: the color of the text in the div

// Methods
```

```
var newElement = document.createElement("p");
newElement.innerHTML = "New paragraph";
document.body.appendChild(newElement);
document.body.removeChild(newElement);
```

Classes (OO-JS):

Classes are a way to define custom objects with their own properties and methods.

```
class Person {
//constructor method
  constructor(name, age) {
    this.name = name;  // Property
    this.age = age;    // Property
  }

  greet() {  // Adding Method
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years
old.`);
  }
}

// Creating an object (instance) of the class
const person1 = new Person('John', 30);
person1.greet();  // Output: Hello, my name is John and I am 30 years old.
```

Error Handling :

Error handling is the process of catching and handling error that occur during the execution of a program.

- Try : A block where you place code that may throw an error. If an error occurs, control is transferred to the `catch` block.
- catch : A block used to handle the error thrown in the `try` block. It receives the error object, which can be used to understand and manage the error.
- final : A block that runs after `try` and `catch`, regardless of whether an error occurred.

```
try {
  // Code that might throw an error
} catch (error) {
  // Error handling code
```

```
} finally {
  // Code that runs regardless of an error
}
```

AJAX :

Ajax allows asynchronous communication with a server without reloading the entire page. It's commonly used with `XMLHttpRequest` to send and receive data from a server.

```
const xhr = new XMLHttpRequest();
xhr.open('GET', 'https://api.example.com/data');
xhr.onload = () => {
  if (xhr.status === 200) {
    console.log(JSON.parse(xhr.responseText));
  }
};
xhr.send();
```

Fetch API:

The Fetch API is a modern, more readable alternative to AJAX for making asynchronous HTTP requests. It uses Promises for handling responses and supports the latest HTTP methods.
he Fetch API sends requests and returns a `Promise` that resolves to the response, allowing easy chaining of `.then()` and `.catch()` for handling success or errors.

```
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) throw new Error('Network response was not ok');
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error('Fetch error:', error));
```

Promises and Asynchronous execution:

Synchronous :

It is sequential. If there is a time consuming task you have to wait for it because it execute line by line.

Asynchronous : It is non-sequential.

Promises:

Promises are a way to handle asynchronous operations in java Script.
In promise there are four properties

1. Status
2. value
3. on fullfillment
4. on rejection
   state of promise - every promise object can have one of 3 status

- pending
- rejected
- fullfilled
  value of promise - initially when promise is created , the state is pending and the value property is undefined.
  after pending it will goes to fullfillment are rejected based on condition.

```
// Example of Promises and Asynchronous execution
var promise = new Promise(function(resolve, reject) {
  setTimeout(function() {
    resolve("Promise resolved.");
  }, 1000);
});

promise.then(function(message) {
  console.log(message); // Output: "Promise resolved."
});
```

JSON:

JSON stands for JavaScript Object Notation. It is a lightweight data format used for exchanging data between a server and a client.

- JSON represents data as key-value pairs, similar to a JavaScript object
- key must be in double quotes
- Data is typically enclosed within `{}` for objects and `[]` for arrays.

```
{
  "name": "Alice",
  "age": 25,
  "isStudent": false,
```

```
  "courses": ["Math", "Science"],
  "address": { "city": "New York", "zip": "10001" }
}
```