JAVA SCRIPT

JavaScript: Why we need JavaScript, What is JavaScript, Environment Setup, Working with Identifiers, Type of Identifiers, Primitive and Non Primitive Data Types, Operators and Types of Operators, Types of Statements, Non - Conditional Statements, Types of Conditional Statements, If and Switch Statements, Types of Loops, Types of Functions, Declaring and Invoking Function, Arrow Function, Function Parameters, Nested Function, Built-in Functions, Variable Scope in Functions, Working With Classes, Creating and Inheriting Classes, In-built Events and Handlers, Working with Objects, Types of Objects, Creating Objects, Combining and cloning Objects using Spread operator, Destructuring Objects, Browser and Document Object Model, Creating Arrays, Destructuring Arrays, Accessing Arrays, Array Methods, Introduction to Asynchronous Programming, Callbacks, Promises, Async and Await, Executing Network Requests using Fetch API, Creating and consuming Modules.

1. Why We Need JavaScript:

JavaScript is an essential language for modern web development because it brings websites to life. Here's why:

- **Interactivity:** JavaScript allows you to create dynamic and engaging user experiences.
    - Imagine a website where you can click buttons, fill out forms, and see immediate responses.
    - JavaScript makes this possible, creating a more interactive and enjoyable experience for users.
- **Dynamic Content:** JavaScript can manipulate the content of a web page after it has been loaded.
    - This means you can update parts of a page without needing to reload the entire thing.
    - For example, displaying search results, changing content based on user actions, or creating animations.
- **Client-Side Scripting:** JavaScript runs directly in the user's web browser, reducing the load on the server.
    - This makes websites faster and more responsive.

- **Versatility:** JavaScript is not limited to just web browsers.
    - It can also be used to build server-side applications (with Node.js), mobile apps, and even desktop applications.

Here are some specific examples of what JavaScript can do:

- **Create interactive forms:** Validate user input, provide real-time feedback, and submit data.
- **Build dynamic user interfaces:** Create dropdown menus, modal windows, and other interactive elements.
- **Handle events:** Respond to user actions like mouse clicks, key presses, and page scrolling.
- **Make websites more visually appealing:** Create animations, transitions, and other visual effects.
- **Develop complex web applications:** Build single-page applications (SPAs) and other advanced web applications.

2 What is JavaScript?

JavaScript is a programming language that developers use to make interactive webpages. From refreshing social media feeds to displaying animations and interactive maps, JavaScript functions can improve a website's user experience. As a client-side scripting language, it is one of the core technologies of the World Wide Web. For example, when browsing the internet, anytime you see an image Scrolling, a click-to-show dropdown menu, or dynamically changing element colors on a webpage, you see the effects of JavaScript.

2.1 How does Java Script

All programming languages work by translating English-like syntax into machine code, which the operating system then runs. JavaScript is broadly categorized as a scripting language, or an interpreted language. JavaScript code is interpreted—that is, directly translated into underlying machine language code by a JavaScript engine.

2.2 Java script Engine

A JavaScript engine is a computer program that runs JavaScript code. The first JavaScript engines were mere interpreters, but all modern engines use just-in-time or runtime compilation to improve performance.

2.3 Client-side Java Script

Client-side JavaScript refers to the way JavaScript works in your browser. In this case, the JavaScript engine is inside the browser code. All major web browsers come with their own built-in JavaScript engines.

Web application developers write JavaScript code with different functions associated with various events, such as a mouse click or mouse hover. These functions make changes to the HTML and CSS.

Here is an overview of how client-side JavaScript works:

1. The browser loads a webpage when you visit it.

2. During loading, the browser converts the page and all its elements, such as buttons, labels, and dropdown boxes, into a data structure called the Document Object Model (DOM).

3. The browser's JavaScript engine converts the JavaScript code into bytecode. This code is an intermediary between the JavaScript syntax and the machine.

4. Different events, such as a mouse click on a button, trigger the execution of the associated JavaScript code block. The engine then interprets the bytecode and makes changes to the DOM.

5. The browser displays the new DOM.

2.4 Server-side JavaScript

Server-side JavaScript refers to the use of the coding language in back-end server logic. In this case, the JavaScript engine sits directly on the server. A server-side JavaScript function can access the database, perform different logical operations, and respond to various events triggered by the server's operating system. The primary advantage of server-side scripting is that you can highly customize the website response based on your requirements, your access rights, and the information requests from the website.

3 Java Script Environment Setup

1. Install Node.js and npm

- **Node.js:** This is the runtime environment that allows you to execute JavaScript code outside of a web browser. It also comes bundled with:
  - **npm (Node Package Manager):** This is a package manager for JavaScript. It allows you to easily install, update, and manage third-party libraries and tools for your projects.

Download and Install:

- Visit the official Node.js website (nodejs.org) and download the installer for your operating system (Windows, macOS, or Linux).
- Follow the on-screen instructions to install Node.js. This will typically also install npm along with it.

Verify Installation:

- Open your terminal or command prompt.
- Type node -v and press Enter. You should see the installed Node.js version.
- Type npm -v and press Enter. You should see the installed npm version.

3.1 Java script frameworks:

Front-End Frameworks:

- **React:** A component-based library for building user interfaces, known for its flexibility and large ecosystem.
- **Angular:** A comprehensive framework for building dynamic web applications, favored for its robust features and enterprise-level support.
- **Vue.js:** A progressive framework that balances flexibility and opinionated structure, making it easy to learn and integrate.

Back-End Frameworks:

- **Node.js:** A runtime environment that allows you to execute JavaScript on the server-side, enabling the creation of scalable and high-performance web servers.
- **Express.js:** A minimal and flexible Node.js framework for building web applications and APIs.
- **NextJS:** A progressive Node.js framework for building efficient, scalable, and enterprise-grade server-side applications

4 Identifiers in JavaScript

In JavaScript, identifiers are used to name variables, functions, and other entities within your code. Here's a breakdown of the rules and best practices:

Rules for Valid Identifiers

- **Case-Sensitive:** JavaScript is case-sensitive. myVariable and myvariable are considered different identifiers.
- **Start with a Letter, Underscore, or Dollar Sign:**
  - myVariable, _myVariable, $myVariable are valid.
  - 123variable is invalid.
- **Consist of Letters, Digits, Underscores, and Dollar Signs:**
  - myVariable123, _myVariable, $variable_1 are valid.
  - my-variable (hyphen) is invalid.
- **Reserved Keywords:** You cannot use JavaScript keywords as identifiers (e.g., if, else, for, var, let, const).

Examples

- **Valid Identifiers:**
  - firstName
  - _lastName
  - $amount
  - customerAge
  - isCustomerActive
- **Invalid Identifiers:**
  - 1stName (starts with a number)
  - my-variable (contains a hyphen)
  - function (reserved keyword)

By following these guidelines, you can write more readable, maintainable, and less error-prone JavaScript code.

Types of Identifiers

In JavaScript, identifiers are names used to identify variables, functions, classes, and other objects. Here's a breakdown of the types of identifiers:

1. Variable Identifiers:

- These are the names given to variables that store data.
- Example: let age = 25; (Here, age is the variable identifier)

2. Function Identifiers:

- These are the names given to functions, which are reusable blocks of code.
- Example: function calculateArea(width, height) { ... } (Here, calculateArea is the function identifier)

3. Class Identifiers:

- These are the names given to classes, which are blueprints for creating objects.
- Example: class Person { ... } (Here, Person is the class identifier)

4. Object Property Identifiers:

- These are the names used to identify properties within an object.
- Example: const person = { name: "Alice", age: 30 }; (Here, name and age are object property identifiers)

Example Program of JavaScript

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script>
        function change_color()
        {
    var element = document.getElementById("btag");
    element.style.color="red";
        }

        function show_userinput()
        {
            var user_name = document.getElementById("txtusername").value;
            alert(user_name);
        }
    </script>
</head>
<body>
    <p id="btag">ACET CSE</p>
    <input type="submit" onclick="change_color();"/>
```

```
    <br>
    <input type="text" id="btuserinput"/>
    <input type="submit" onclick="show_userinput()"/>

</body>
</html>
```

Output :

ACET CSE

Submit

Submit

Before click submit button

5 Primitive and Non Primitive types

Primitive Data Types :

- **Definition:** These are the basic building blocks of data in programming languages. They represent single values of a specific type.

Non-Primitive Data Types (Reference Types)

- **Definition:** These are more complex data structures that can hold multiple values or collections of values.

Primitive Data Types:

- **Number:** Represents numerical values (e.g., 10, 3.14, -5, Infinity, NaN).
- **String:** Represents a sequence of characters (e.g., "Hello", 'world', '').
- **Boolean:** Represents a logical entity (either true or false).
- **Null:** Represents the intentional absence of any value (e.g., null).

- **Undefined:** Represents a variable that has been declared but has not been assigned a value (e.g., let myVar;).
- **Symbol:** Represents a unique and immutable value (introduced in ES6).

Non-Primitive Data Types (Objects):

- **Object:** The most fundamental building block for creating more complex data structures. Can hold a collection of key-value pairs.
- **Array:** An ordered collection of values (can be of different data types).
- **Function:** A block of code designed to perform a specific task.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script>
    //
    let a=10;
    let b= "cat";
    let d=true;
    if(d)
    {
        document.writeln("True");
    }
    else
    {
    document.writeln("False");
    }
    //objects
    const student = {number:500,name:"abc",dept:"cse"};
    document.writeln( "</br> student name:" + student.name)
    // Array
    let student = [500,"abc","cse"];
    document.writeln("</br>" + student[0]);
    // function
    function test_function()
    {
    document.getElementById("demo").innerHTML="</br>" + "function is working";
    }
</script>
</head>
```

```
<body>
</br>
    <input type="submit" onclick="test_function();" value="function" />
    <p id="demo">
    </p>
</body>
</html>
```

Output

True
Student name: ally
50$
function

Function is working

6 Operators

In programming, operators are symbols or keywords that perform specific operations on one or more values (called operands). They are essential for manipulating data and controlling the flow of a program.

Types of Operators:

Here are some common types of operators:

1. **Arithmetic Operators:**
   - Perform basic mathematical operations.
   - Examples: + (addition), - (subtraction), * (multiplication), / (division), % (modulus - remainder after division)
2. **Relational Operators:**
   - Compare values and return a boolean result (true or false).
   - Examples: == (equal to), != (not equal to), < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to)
3. **Logical Operators:**
   - Combine boolean expressions.
   - Examples: && (AND), || (OR), ! (NOT)
4. **Bitwise Operators:**
   - Operate on individual bits of data.
   - Examples: & (bitwise AND), | (bitwise OR), ^ (bitwise XOR), ~ (bitwise NOT), << (left shift), >> (right shift)
5. **Assignment Operators:**
   - Assign values to variables.
   - Examples: = (simple assignment), += (add and assign), -= (subtract and assign), *= (multiply and assign), /= (divide and assign), %= (modulus and assign)
6. **Increment/Decrement Operators:**

- Increase or decrease the value of a variable by 1.
- Examples: ++ (increment), -- (decrement)

7. **Conditional (Ternary) Operator:**
   - A shorthand way to write an if-else statement.
   - Example: condition ? value_if_true : value_if_false

Example Program:

```
x = 10
y = 5

# Arithmetic operators:
sum = x + y  # sum will be 15
difference = x - y  # difference will be 5
product = x * y  # product will be 50
quotient = x / y  # quotient will be 2.0
remainder = x % y  # remainder will be 0

# Relational operators:
is_equal = (x == y)  # is_equal will be False
is_not_equal = (x != y)  # is_not_equal will be True
is_greater = (x > y)  # is_greater will be True

# Logical operators:
both_true = (x > 0 and y > 0)  # both_true will be True
either_true = (x > 0 or y > 0)  # either_true will be True
not_true = not (x == y)  # not_true will be True

# Assignment operators:
x += 5  # x will now be 15
y -= 2  # y will now be 3

# Increment/decrement operators:
x++  # x will now be 16
y--  # y will now be 2

# Conditional operator:
result = "x is greater" if x > y else "y is greater"
```

7 Conditional Statements

Conditional statements in JavaScript allow you to control the flow of your program based on specific conditions. They enable you to execute different blocks of code depending on whether a certain condition is true or false. Here are the primary conditional statements:

If : Executes a block of code only if a specific condition is true :

```
if (condition) {

  // Code to be executed if the condition is true

}
```

Example on If :

```
let age = 25;

if (age >= 18) {

  console.log("You are an adult.");

}
```

If ..... Else : Executes one block of code if the condition is true, and another block if it's false.

```
if (condition) {

  // Code to be executed if the condition is true

} else {

  // Code to be executed if the condition is false

}
```

If....else Example

```
let isRaining = true;


if (isRaining) {

  console.log("Take an umbrella.");

} else {

  console.log("Enjoy the sunshine!");}
```

`if...else if...else` statement: Handles multiple conditions sequentially. If the first condition is true, its block executes. If not, the next else if condition is checked, and so on. If none of the conditions are true, the else block (if present) executes.

```
if (condition1) {
  // Code to be executed if condition1 is true
} else if (condition2) {
  // Code to be executed if condition1 is false and condition2 is true
} else {
  // Code to be executed if neither condition1 nor condition2 is true
}
```

`if...else if...else` statement example:

```
let score = 85;
if (score >= 90) {
  console.log("Excellent!");
} else if (score >= 80) {
  console.log("Great job!");
} else {
  console.log("Good effort.");
}
```

Switch statement: Evaluates an expression and compares it to multiple cases. If a match is found, the corresponding block of code executes.

```
switch (expression) {
  case value1:
    // Code to execute if expression matches value1
    break;
  case value2:
    // Code to execute if expression matches value2
    break;
  // ... more cases
  default:
    // Code to execute if none of the cases match}
```

Switch Example

```
let dayOfWeek = "Sunday";
switch (dayOfWeek) {
  case "Saturday":
  case "Sunday":
    console.log("Weekend!");
    break;
  case "Monday":
  case "Tuesday":
  case "Wednesday":
  case "Thursday":
  case "Friday":
    console.log("Weekday.");
    break;
  default:
    console.log("Invalid day.");
}
```

8 Types loops:

- for - loops through a block of code a number of times
- for/in - loops through the properties of an object
- for/of - loops through the values of an iterable object
- while - loops through a block of code while a specified condition is true
- do/while - also loops through a block of code while a specified condition is true

For : The for statement creates a loop with 3 optional expressions:

1. **Expression 1** is executed (one time) before the execution of the code block.
2. **Expression 2** defines the condition for executing the code block.
3. **Expression 3** is executed (every time) after the code block has been executed.

Example program:

```
<!doctype html>
<html>
<body>

<h2>JavaScript Program for First 10 Numbers:</h2>

<p id="demo"></p>

 <script>
   let text= " ";
   for(let i=0; i<5; i++)
   {
    text+="The number is " + i + "<br>";
   }
   document.getElementById("demo").innerHTML=text;

 </script>

</body>
</html>
```

Output:

JavaScript For Loop

The number is 0
The number is 1
The number is 2
The number is 3
The number is 4

For-in: The JavaScript for in statement loops through the properties of an Object:

- The **for in** loop iterates over a **student** object
- Each iteration returns a **key** (x)
- The key is used to access the **value** of the key
- The value of the key is student[x]

```
const student = {name:"John", dept:"cse", age:25};
let obj= "";
for (let x in student) {
obj += student[x] + " " + "<br>" ;}
document.getElementById("object").innerHTML = obj;
```

oupt put:

For-of:

The JavaScript for of statement loops through the values of an iterable object.

It lets you loop over iterable data structures such as Arrays, Strings, Maps, NodeLists, and more:

```
let person = [101,"ravi",5000];
let arr="";
for ( let y of person)
{

  arr += y  + "<br>";

  document.getElementById("array").innerHTML = arr;
}
```

Out Put :

101
ravi
5000

9 Types of Functions

JavaScript functions are used to perform operations. We can call JavaScript function many times to reuse the code.

Advantage of JavaScript function

Functions are useful in organizing the different parts of a script into the several tasks that must be completed. There are mainly two advantages of JavaScript functions.

1. **Code reusability:** We can call a function several times in a script to perform their tasks so it saves coding.
2. **Less coding:** It makes our program compact. We don't need to write many lines of code each time to perform a common task.

Types of Functions in JavaScript:

1. **Named Functions**
2. **Anonymous functions**
3. **Arrow functions**
4. **Generator functions**
5. **Async Functions**

1. Named Functions :
   - Function name is explicitly declared.
   - can be called before they are defined (hoisted).

Example :

```
<!DOCTYPE Html>

<html lang="en">
<head>
    <title>functions</title>
<script>
function myFunction(name)
{
   document.writeln( "Hello, " + name + "!");
}
</script>
</head>
<body>
<button onclick="myFunction('Harry Potter')">try it</button>
</body>
</html>
```

2. **Anonymous Functions**

An anonymous function is simply a function that does **not have a name**. Unlike named functions, which are declared with a name for easy reference, anonymous functions are usually created for specific tasks and are often assigned to variables or used as arguments for other functions.

commonly used in various scenarios, such as callbacks, event handlers, and functional programming tasks.

Example :

```
const greet = function () {
   console.log ("Welcome to GeeksforGeeks!");
};
```

Example of callback

```
    setTimeout(function() {

   console.log("This message appears after 2 seconds");
}, 2000);
```

3. **Arrow functions**

An **arrow function expression** is a compact alternative to a traditional function expression, with some semantic differences and deliberate limitations in usage:

Arrow functions without parameters

An arrow function without parameters is defined using empty parentheses **()**. This is useful when you need a function that doesn't require any arguments.

```
const arrow = () => {
   console.log( "Arrow function" );
}

arrow();
```

Arrow function with single parameter:

If your arrow function has a single parameter, you can omit the parentheses around it.

```
const square =( x) => x*x;
console.log(square(4));
```

Arrow function with multiple parameters:

Arrow functions with multiple parameters, like **(param1, param2) => { }**, simplify writing concise function expressions in JavaScript, useful for functions requiring more than one argument.

```
const arr = ( x, y, z ) => {
    console.log( x + y + z )
}

arr( 10, 20, 30 );
```

Arrow functions with array :

```
const subjects = ["MEANSTACK","HL","DMS"]

console.log(subjects.map((subject) => subject.length));
```

o/p: [9, 2, 3]

4. Generator Functions:

Generator functions are a special types of functions in JavaScript, introduced in ES6, that have the built-in capability to be **paused and resumed** allowing us to take control of the execution flow and generate multiple values.

The syntax, as shown below, is pretty much similar with regular functions apart from the new *function** keyword.

```
function* someGeneratorFunction() {
  console.log("Start of the function");
  yield 1;
  console.log("Middle of the function");
  yield 2;
  console.log("End of the function");
}
const generator = someGeneratorFunction(); // Returns generator object
console.log(generator.next().value);
```

5. Async Functions in JavaScript

- Functions designed to handle asynchronous operations (like fetching data from a server, reading/writing files).
- Use the async keyword before the function declaration.

Key Features:

- **await keyword:** Used within an async function to pause execution until a Promise resolves.
- **Cleaner Code:** Makes asynchronous code look more synchronous and easier to read.
- **Error Handling:** Can easily handle errors using try...catch blocks.

What is Asynchronous:

In web development, asynchronous programming is a fundamental concept that allows your website or web application to perform multiple tasks concurrently without blocking the main program's execution. This is crucial for creating responsive and efficient user experiences.

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    return data;
  } catch (error) {
    console.error('Error fetching data:', error);
    throw error; // Re-throw the error for further handling
  }
}

fetchData().then(data => { console.log('Data:', data); }).catch(error => {
console.error('Error:', error); });
```

In this example, the fetchData function fetches data from a server asynchronously using async/await. The await keyword pauses the execution of the function until the fetch operation completes, making the code easier to read and write

10 Declaring and Invoking Function

Function Declaration:

- You define a function using the function keyword followed by the function's name.
- Parameters are placeholders for values that will be passed to the function when it's called.
- The function body contains the code that will be executed when the function is invoked.

Function Invoking :

- To call a function, you use its name followed by parentheses.
- Arguments are the actual values that are passed to the function when it's called. These values are assigned to the corresponding parameters within the function's body.

In functions types we see the different types of functions declaration and invoking

11 Arrow Functions

An arrow function is a shorter syntax for writing functions in JavaScript. Introduced in ES6, arrow functions allow for a more concise and readable code, especially in cases of small functions. Unlike regular functions, arrow functions don't have their own this, but instead, inherit it from the surrounding context.

- Arrow functions are written with the => symbol, which makes them compact.
- They don't have their own this. They inherit this from the surrounding context.
- For functions with a single expression, the return is implicit, making the code more concise.
- Arrow functions do not have access to the arguments object, which is available in regular functions.

```
const add = (a, b) => a + b;
console.log(add(5, 5));
```

o/p: 5

1. Arrow Function without Parameters

An arrow function without parameters is defined using empty parentheses (). This is useful when you need a function that doesn't require any arguments.

```
const show = () => {
   console.log( "Acet" );
}

show();
```

o/p: Acet

2. Arrow Function with Single Parameters

If your arrow function has a single parameter, you can omit the parentheses around it.

```
const square = x => x*x;

console.log(square(4));
```

o/p: 16

3. Arrow Function with Multiple Parameters

Arrow functions with multiple parameters, like (**param1, param2**) => { }, simplify writing concise function expressions in JavaScript, useful for functions requiring more than one argument.

```
const show = ( x, y, z ) => {

   console.log( x + y + z )

}

show( 10, 20, 30 );   o/p: 60
```

4. Arrow Function with Default Parameters

Arrow functions support default parameters, allowing predefined values if no argument is passed, making JavaScript function definitions more flexible and concise.

```
const add = ( x, y, z = 30 ) => {

   console.log( x + " " + y + " " + z);

};

add( 10, 20 );
```

o/p: 30

12 Nested functions

In JavaScript, Functions within another function are called "Nested function." These nested functions have access to the variables and parameters of the outer (enclosing) function, creating a scope hierarchy. A function can have one or more inner functions.

Example:

```
function giveMessage(message) {
  let userMsg = message;
  function caller(userName) {
      let name = userName;
      let greet = userMsg + " " + name;
      return greet;
  }
  userMsg = caller("John");
  return userMsg;
}

console.log(giveMessage("The world says hello dear: "));
// The world says hello dear: Bob
```

13 Built-in Functions:

JavaScript provides two-timer built-in functions. Let us explore these timer functions. Below is the table with some of these built-in functions to understand their significance and usage.

Built-in functions	Description	Example
alert()	It throws an alert box and is often used when user interaction is required to decide whether execution should proceed or run.	alert("Let us proceed");
confirm()	It throws a confirm box where user can click "OK" or "Cancel". If "OK" is clicked, the function returns "true", else returns "false".	let decision = confirm("Shall we proceed?");
prompt()	It produces a box where user can enter an input. The user input may be used for some processing later. This function takes parameter of type string which represents the label of your name in the box.	let userInput = prompt("Please enter your name");

isNaN()	This function checks if the data-type of given parameter is number or not. If number, it returns "false", else it returns "true".	isNaN(30); //false
		isNaN("hello"); //true

isFinite()	It determines if the number given as parameter is a finite number. If the parameter value is, NaN, positive infinity, or negative infinity, this method will return false, else will return true.	isFinite(30); //true
		isFinite("hello"); //false

parseInt()	This function parses string and returns an integer number. It takes two parameters. The first parameter is the string to be parsed. The second parameter represents radix which is an	parseInt("10"); //10 parseInt("10, 20, 30"); //10, only the

Example: Using the confirm method with a condition

```
<!doctype html>
<html>
<head>
</head>
<body>

<p>Click the button to invoke the confirm().</p>

<button onclick="myfunction()" Click Here</button>
<p id="conf"></p>
<script>
function myfunction() {
if (window.confirm("Do you really want to delete?")) {
  document.getElementById("conf").innerHTML = "you have successfully deleted
the file"
}
}
</script>
</body>
</html>
```

Output:

Click the button to invoke the confirm().

Click Here

you have successfully deleted the file

14 Variable Scope in functions

Variable declaration in the JavaScript program can be done within the function or outside the function. But the accessibility of the variable to other parts of the same program is decided based on the place of its declaration. This accessibility of a variable is referred to as scope.

JavaScript scopes can be of three types:

- Global scope
- Local scope
- Block scope

1. Global scope

Variables defined outside function have Global Scope and they are accessible anywhere in the program.

Example:

```
//Global variable

var greet = "Hello JavaScript";

function message()

{

    //Global variable accessed inside the function

    console.log("Message from inside the function: " + greet);

}

message();

//Global variable accessed outside the function

console.log("Message from outside the function: " + greet);

//Message from inside the function: Hello JavaScript

//Message from outside the function: Hello JavaScript
```

2. local scope

Variables declared inside the function would have local scope. These variables cannot be accessed outside the declared function block.

```
function message() {

//Local variable

var greet = "Hello JavaScript";

//Local variables are accessible inside the function
```

console.log("Message from inside the function: " + greet);

)

message();

//Local variable cannot be accessed outside the function

console.log("Message from outside the function: " + greet);

//Message from inside the function: Hello JavaScript

//Uncaught ReferenceError: greet is not defined

3. Block scope

In 2015, JavaScript introduced two new keywords to declare variables: let and const. Variables declared with 'var' keyword are function-scoped whereas variables declared with 'let' and 'const' are block-scoped and they exist only in the block in which they are defined.

Consider the below example:

```
1. function testVar() {
2.   if (10 == 10) {
3.         var flag = "true";
4.   }
5.   console.log(flag); //true
6. }
7.

8. testVar();
```

In the above example, the variable flag declared inside 'if' block is accessible outside the block since it has function scope.

Modifying the code to use 'let' variable will result in an error:

```
1. function testVar() {
2.     if (10 == 10) {
3.         let flag = "true";
4.     }
5.     console.log(flag); // ReferenceError: flag is not defined
6. }
7.

8. testVar();
```

The usage of 'let' in the above code snippet has restricted the variable scope only to 'if' block.

'const' has the same scope as that of 'let' i.e., block scope.

15 Working with classes

Classes in JavaScript are a blueprint for creating objects, introduced in ES6. They encapsulate data and behaviour by defining properties and methods, enabling object-oriented programming. Classes simplify the creation of objects and inheritance, making code more organized and reusable.

Example of class:

```
class emp{
    constructor(name,age){
     this.name= name;
     this.age = age;}
}
let obj = new emp('veeraldra',35);
console.log(obj.age);
console.log(obj.emp);
```

The constructor method is a special method:

- It has to have the exact name "constructor"
- It is executed automatically when a new object is created.
- It is used to initialize object properties.

Inheriting Classes

To create a class inheritance, use the extends keyword.

A class created with a class inheritance inherits all the methods from another class:

```
class car{
    constructor(brand)
    {
        this.carname = brand;
    }
    present(){
        return 'I have a' + this.carname;
    }
}
class model extends car{
    constructor(brand,mod){
        super(brand);
        this.model=mod;
    }
    show()
    {
        return this.present() + ', it is a' + this.model;
    }
}
```

```
let myCar = new model('red', 'Ford');
let a= myCar.show();
console.log(a);
```

In-built Events and Handlers:

The change in the state of an object is known as an **Event**. In html, there are various events which represents that some activity is performed by the user or by the browser. When javascript code is included in HTML, js react over these events and allow the execution. This process of reacting over the events is called **Event Handling**. Thus, js handles the HTML events via **Event Handlers**.

Below are some of the built-in event handlers.

Event	Event-handler	Description
click	onclick	When the user clicks on an element, the event handler onclick handles it.
keypress	onkeypress	When the user presses the keyboard's key, event handler onkeypress handles it.
keyup	onkeyup	When the user releases the keyboard's key, the event handler onkeyup handles it.
load	onload	When HTML document is ready in the browser, event handler onload handles it
blur	onblur	When an element loses focus, the event handler onblur handles it.
change	onchange	When the selected, checked state changes for input, select or text area element changes, event handler onchange handles it.

Or 1.Mouse-events:

Event Performed	Event Handler	Description
click	onclick	When mouse click on an element
mouseover	onmouseover	When the cursor of the mouse comes over the element
mouseout	onmouseout	When the cursor of the mouse leaves an element
mousedown	onmousedown	When the mouse button is pressed over the element
mouseup	onmouseup	When the mouse button is released over the element
mousemove	onmousemove	When the mouse movement takes place.

2.Keyboard events:

Event Performed	Event Handler	Description
Keydown & Keyup	onkeydown & onkeyup	When the user press and then release the key.

3.Form events:

Event Performed	Event Handler	Description
focus	onfocus	When the user focuses on an element
submit	onsubmit	When the user submits the form
blur	onblur	When the focus is away from a form element
change	onchange	When the user modifies or changes the value of a form element

4.Window/Document events

Event Performed	Event Handler	Description
load	onload	When the browser finishes the loading of the page
unload	onunload	When the visitor leaves the current webpage, the browser unloads it
resize	onresize	When the visitor resizes the window of the browser

Click Event example:

```
<html>

<head> Javascript Events </head>

<body>

<script language="Javascript" type="text/Javascript">

        <!--

        function clickevent()

        {

                document.write("This is JavaTpoint");

        }

        //-->

</script>

<form>

<input type="button" onclick="clickevent()" value="Who's this?"/>

</form>

</body>

</html>
```

output:

Javascript Events

Who's this?

on click this button its display the message :

This is JavaTpoint

Working with Objects:

In any programming language when real-world entities are to be coded, then variables are used. For most of the scenarios, a variable to hold data that represents the collection of properties is required.

For instance, to create an online portal for the car industry, Car as an entity must be modelled so that it can hold a group of properties.

Such type of variable in JavaScript is called an Object. An object consists of state and behavior.

The State of an entity represents properties that can be modeled as key-value pairs.

The Behavior of an entity represents the observable effect of an operation performed on it and is modelled using functions.

Example:

A Car is an object in the real world.

State of Car object:

- Coloured
- Model = VXI
- Current gear = 3
- Current speed = 45 km/ hr
- Number of doors = 4
- Seating Capacity = 5

The behavior of Car object:

- Accelerate
- Change gear
- Brake

Types of Objects:

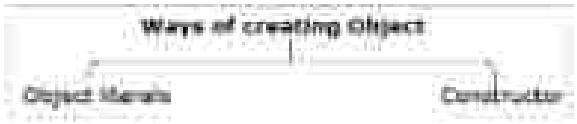
JavaScript objects are categorized as follows:

Creating Objects:

In JavaScript objects, the state and behaviour is represented as a collection of properties.

Each property is a [key-value] pair where the key is a string and the value can be any JavaScript primitive type value, an object, or even a function.

JavaScript objects can be created using two different approaches.



1. JavaScript Object by object literal:

The syntax of creating object using object literal is given below:

1. object={property1:value1,property2:value2..,property(N):value(N)}

As you can see, property and value is separated by : (colon).

Let's see the simple example of creating object in javascript.

```
1. <script>
2. emp={id:102,name:"Shyam Kumar",salary:40000}
3. document.write(emp.id+" "+emp.name+" "+emp.salary);
4. </script>
```

Output of the above example

```
102 Shyam Kumar 40000
```

The syntax of creating object directly is given below:

var objectname= new Object();

Here, **new keyword** is used to create object.

Let's see the example of creating object directly.

```
<script>
var emp=new Object();
emp.id=101;
emp.name="Ravi Malik";
emp.salary=50000;
document.write(emp.id+" "+emp.name+" "+emp.salary);
</script>
```

Output of the above example

```
101 Ravi 50000
```

2.By using an Object constructor

Here you need to create function with arguments. Each argument value can be assigned in the current object by using this keyword.

The **this keyword** refers to the current object.

The example of creating object by object constructor is given below.

```
1. <script>
2. function emp(id,name,salary){
3. this.id=id;
4. this.name=name;
5. this.salary=salary;
6. }

7. e=new emp(100,"Vimal Jaiswal",10000);
8.
9. document.write(e.id+" "+e.name+" "+e.salary);
10. </script>
```

Output of the above example

```
101 Vimal Jaiswal 30000
```

Combining and cloning Objects using Spread operator:

The spread operator is used to combine two or more objects. The newly created object will hold all the properties of the merged objects.

Syntax:

```
let object1Name = {
  //properties
};
let object2Name = {
  //properties
};
let combinedObjectName = {
  ...object1Name,
  ...object2Name
};
//the combined object will have all the properties of object1 and object2
```

Destructuring Objects:

When destructuring the objects, we use keys as the name of the variable. The variable name must match the property (or keys) name of the object. If it does not match, then it receives an **undefined** value. This is how JavaScript knows which property of the object we want to assign.

In object destructuring, the values are extracted by the keys instead of position (or index).

First, try to understand the basic assignment in object destructuring by using the following example.

Example - Simple assignment

```
1. const num = {x: 100, y: 200};
2. const {x, y} = num;
3.
4. console.log(x); // 100
5. console.log(y); // 200
```

Example - Basic Object destructuring assignment

```
1. const student = {name: 'Arun', position: 'First', rollno: '24'};
2. const {name, position, rollno} = student;
3. console.log(name); // Arun
4. console.log(position); // First
5. console.log(rollno); // 24
```

Object destructuring in functions

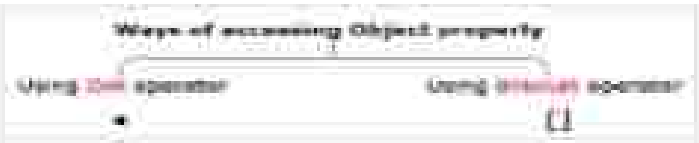
```
1. let myObject = { name: 'Marty', age: 65, country: 'California' };
2. function showDetails({ country }) {
3.     console.log(country);
4. }
```

Accessing Object Properties.

After the object has been created, its variables or methods can be accessed in two different ways.

Using:

- dot operator
- bracket operator



Syntax:

For retrieving simple/literal values.

```
1. object.Property_Name;
2. // OR
3. object["Property_Name"];
4.
```

For setting state/behavior value,

```
1. object[dynamic_key] = value
2. //OR
3. object.name[key] = value
```

To work with all the keys of an object, there is a particular form of the loop: for in. This is a different way from the for() construct.

Syntax:

```
1. for (key in object) {
2.   // execute an iteration for each key-value pair in object
3. }
```

Example:

```
1. let user = {
2.   name: "Rusty",
3.   age: 14,
4.   isCool: false
5. };
6.
7. for (let key in user) {
8.   // keys
9.   console.log(key); // name, age, isCool
10.   // values for the keys
11.   console.log(user[key]); // Rusty, 14, true
12. }
```

In case of "for" constructs, it allows to declare the looping variable inside the loop, like let key here.

Also, another variable name can be used instead of key. For example, "for (let prop in obj)" is also commonly used.

Built-In Global Objects:

The built-in JavaScript object 'Date' allows us to work with dates and times displayed as part of the web page. It can be instantiated wherever required using one of the many constructors available.

Example:

```
1. let dateObject1 = new Date();
2. console.log("Date is: " + dateObject1);
3. //OUTPUT: Date is: Thu Jun 18, 2020, 22:17:36 GMT+0530 (India Standard Time)
```

OR

```
1. let dateObject2 = new Date(2020, 5, 18, 22, 20, 23, 0000);
2. console.log("Date is: " + dateObject2);
3. //OUTPUT: Date is: Thu Jun 18, 2020, 22:20:23 GMT+0530 (India Standard Time)
```

After the object of type 'Date' is ready, you can access and use the built-in methods. Most of the methods provided by the object 'Date' aim at getting a specific portion of the current time.

The below table is about the getter methods available on object Date.

Method	Description
getDate()	Returns the numeric day of the month. The value ranges from 1 to 31.
getDay()	Returns numeric day of week. Value ranges from 0 to 6.
getFullYear()	Returns four digit year (YYYY).
getHours()	Returns numeric hour. Value ranges from 0 to 23.
getMonth()	Returns numeric month. Value ranges from 0 to 11.
getMilliseconds()	Returns numeric milliseconds. Value ranges from 0 to 999.
getTime()	Returns number of milliseconds since 1/1/1970 at 12 a.m.

Example:

```
1. let dateObject1 = new Date();
2. console.log("Date is: " + dateObject1.getDate());
3. console.log("Day is: " + dateObject1.getDay());
4. console.log("Year is: " + dateObject1.getYear());
5. console.log("Hours: " + dateObject1.getHours());
6. console.log("Month is: " + dateObject1.getMonth());
7. console.log("Time is: " + dateObject1.getTime());
8. console.log("Millisecond: " + dateObject1.getMilliseconds());
9. /*
10. OUTPUT:
11. Date is: 18


12. Day is: 4
13. Year is: 120
14. Hours: 22
15. Month is: 5
16. Time is: 1592499518512
17. Millisecond: 512
18. */
```

Setter methods available on object Date are listed below:

Method	Description
setDate()	Sets the numeric day of the month. Value range from 1 to 31.
setFullYear()	Sets four-digit year (YYYY).
setHours()	Sets numeric hour. The value ranges from 0 to 23.
setMonth()	Sets numeric month. The value ranges from 0 to 11.
setMilliseconds()	Sets numeric milliseconds. The value ranges from 0 to 999.
setTime()	Sets the number of milliseconds from 1/1/1970 at 12 a.m.

Example:

```
1. let dateObject = new Date();
2. dateObject.setDate(1);
3. dateObject.setYear(1996);
4. dateObject.setMonth(8);
5. dateObject.setHours(7);
6. dateObject.setMilliseconds(2000);
7.
8. console.log("Date is:" + dateObject.getDate());
9. console.log("Year is:" + dateObject.getYear());
10. console.log("Month:" + dateObject.getMonth());
11. console.log("Hours:" + dateObject.getHours());
12. console.log("Milliseconds:" + dateObject.getMilliseconds());
13. /*
14. Output:
15. Date is:1
16. Year is:96
17. Month:8
18. Hours:7
19. Milliseconds:0
20. */
```

charAt()

It retrieves a character that resides on the index passed as an argument.

Example:

```
1. let myString = "Hello World";
2. console.log("Character at position 4 is:" + myString.charAt(4));
3. //Returns: Character is possible 4 is:o
```

concat()

It accepts an indefinite number of string arguments, joins them, and returns the combined result as a new string.

Example:

```
1. let myStr1 = "Hello";
2. let myStr2 = " ";
3. let myStr3 = "World";
4. console.log("Concatenated string:" + myStr1.concat(myStr2,myStr3));
5. //Returns: Concatenated string:Hello World
```

indexOf()

It returns the index of the given character or maybe the given set of characters in a string passed as an argument.

Example:

```
1. let myString = "Hello World";
2. console.log("Index of character e is:" + myString.indexOf("e"));
3. //Returns: Index of character 1 is:1
```

substr()

It is like the substring() method.

The difference is, if the second parameter is provided, it takes the first parameter as start Index and second parameter as length for slicing string.

Example:

```
1. var myString="HelloWorld";
2. console.log("Substr using 2 parameters:"+myString.substr(2,5));
3. console.log("Substr using 1 parameter:"+myString.substr(5));
4. //Returns:
5. Substr using 2 parameters:llowW
6. Substr using 1 parameter:World
```

toLowerCase()

Converts characters in string to lowercase

Example:

```
1. let myString="HelloWorld";
2. console.log("Lowercase string:"+myString.toLowerCase());
3. //Returns Lowercase string:helloworld
```

toUpperCase()

Converts characters in string to uppercase

Example:

```
1. let myString="HelloWorld";
2. console.log("Uppercase string:"+myString.toUpperCase());
3. //Returns Uppercase string:HELLOWORLD
```

Browser Object Model:

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

The default object of browser is window means you can call all the functions of window by specifying window or directly. For example:

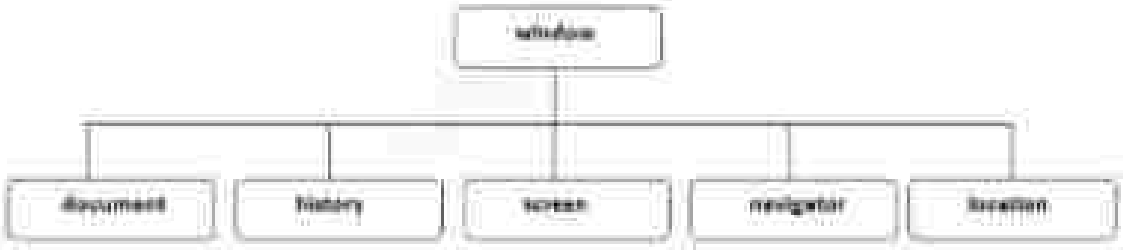
```
1. window.alert("hello javatpoint");
```

It same as:

```
1. alert("hello javatpoint");
```

You can use a list of properties of other objects defined underneath the window object like document, history, screen, navigator, location, innerHeight, innerWidth.

Note: The document object represents the html document. It forms DOM (Document Object Model).



Window Object

The **window object** represents a window in browser. An object of window is created automatically by the browser.

Window is the object of browser, **it is not the object of javascript**. The javascript objects are string, array, date etc.

Methods of window object

The important methods of window object are as follows:

Method	Description
alert()	displays the alert box containing message with ok button.
confirm()	displays the confirm dialog box containing message with ok and cancel button.
prompt()	displays a dialog box to get input from the user.
open()	opens the new window.
close()	closes the current window.

setTimeout() performs action after specified time like calling function, evaluating expressions etc.

JavaScript History Object

The **JavaScript history object** represents an array of URLs visited by the user. By using this object, you can load previous, forward or any particular page.

The history object is the window property, so it can be accessed by:

1. window.history

Or,

1. history

Methods of JavaScript history object

There are only 3 methods of history object.

No.	Method	Description
1	forward()	loads the next page
2	back()	loads the previous page.
3	go()	loads the given page number

JavaScript Navigator Object

The **JavaScript navigator object** is used for browser detection.

It can be used to get browser information such as appName, appCodeName, userAgent etc.

The navigator object is the window property, so it can be accessed by:

1. window.navigator

Or,

1. navigator

Property of JavaScript navigator object

No.	Property	Description
1	appName	returns the name
2	appVersion	returns the version
3	appCodeName	returns the code name
4	cookieEnabled	returns true if cookie is enabled otherwise false
5	userAgent	returns the user agent
6	language	returns the language. It is supported in Netscape and Firefox only.
7	userLanguage	returns the user language. It is supported in IE only.
8	plugins	returns the plugins. It is supported in Netscape and Firefox only.
9	systemLanguage	returns the system language. It is supported in IE only.
10	mimeTypes[]	returns the array of mime type. It is supported in Netscape and Firefox only.
11	platform	returns the platform e.g. Win32.
12	online	returns true if browser is online otherwise false.

JavaScript Screen Object

The **JavaScript screen object** holds information of browser screen. It can be used to display screen width, height, colorDepth, pixelDepth etc.

The navigator object is the window property, so it can be accessed by:

```
1. window.screen
```

Or,

```
1. screen
```

Property of JavaScript Screen Object

There are many properties of screen object that returns information of the browser.

No.	Property	Description
1	width	returns the width of the screen
2	height	returns the height of the screen
3	availWidth	returns the available width
4	availHeight	returns the available height
5	colorDepth	returns the color depth
6	pixelDepth	returns the pixel depth.

Document Object Model :

The **document object** represents the whole html document.

When html document is loaded in the browser, it becomes a document object. It is the **root element** that represents the html document. It has properties and methods. By the help of document object, we can add dynamic content to our web page.

As mentioned earlier, it is the object of window. So

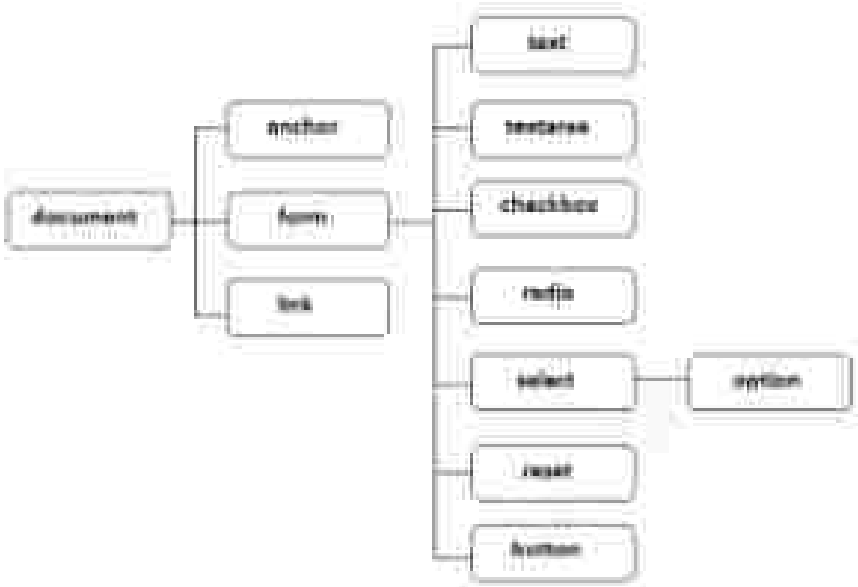
```
1. window.document
```

Is same as

```
1. document;
```

Properties of document object

Let's see the properties of document object that can be accessed and modified by the document object.



Methods of document object:

We can access and change the contents of document by its methods.

The important methods of document object are as follows:

Method	Description
write("string")	writes the given string on the document.
writeln("string")	writes the given string on the document with newline character at the end.
getElementById()	returns the element having the given id value.
getElementsByName()	returns all the elements having the given name value.
getElementsByTagName()	returns all the elements having the given tag name.
getElementsByClassName()	returns all the elements having the given class name....

Accessing field value by document object

In this example, we are going to get the value of input text by user. Here, we are using **document.form1.name.value** to get the value of name field.

Here, **document** is the root element that represents the html document.

form1 is the name of the form.

name is the attribute name of the input text.

value is the property, that returns the value of the input text.

Let's see the simple example of document object that prints name with welcome message.

```
1. <script type="text/javascript">
2. function printvalue(){
3. var name=document.form1.name.value;
4. alert("Welcome: "+name)
5. }
6. </script>
7.
8. <form name="form1">
9. Enter Name:<input type="text" name="name"/>
10. <input type="button" onclick="printvalue()" value="print name"/>
11. </form>
```

Output of the above example

Enter Name: Invite

The alert box will be displayed welcome Invite ..

Working with Arrays:

Objects in JavaScript is a collection of properties stored as key-value pair.

Often, there is requirement for an ordered collection, where there are 1st, 2nd, 3rd element, and so on. For example, you need to store a list of students in a class based on their roll numbers

It is not convenient to use an object here, because objects don't store the values in an ordered fashion. Also, a new property or element cannot be inserted "between" the existing ones.

This is why Array is used to store values in order.

Array in JavaScript is an object that allows storing multiple values in a single variable. An array can store values of any datatype. An array's length can change at any time, and data can be stored at non-contiguous locations in the array.

Example:

```
let numArr = [1, 2, 3, 4];
let empArr = ["Johnson", 105678, "Chicago"];
```

The elements of the array are accessed using an index position that starts from 0 and ends with the value equal to the length of the array minus 1.

Examples :

```
1. let numArr = [1, 2, 3, 4];
2. console.log(numArr[0]); //1
3. console.log(numArr[3]); //4
```

Creating Arrays:

Arrays can be created using the literal notation or array constructor.

Array Literal Notation:

Arrays are created using literal notation almost all the time.

Syntax:

```
1. let myArray = [element1, element2, ..., elementN];
```

Example:

```
1. let colors = ["Red", "Orange", "Green"]
```

Array Constructor:

Arrays can be created using the Array constructor with a single parameter which denotes the array length. The parameter should be an integer between 0 and 232-1 (inclusive). This creates empty slots for the array elements. If the argument is any other number, a RangeError exception is thrown.

Syntax:

```
1. let myArray = new Array(arrayLength);
```

Example:

```
1. let colors = new Array(2);
2. console.log(colors.length); //2
3.
4. //Assign values to an empty array using indexes
5. colors[0] = "Red";
6. colors[1] = "Green";
7. console.log(colors); //['Red','Green']
```

If more than one argument is passed to the Array constructor, a new Array with the given element(s) is created.

Syntax:

```
1. let myArray = new Array(element1, [, element2, ..., elementN]);
```

Example:

```
1. let colors = new Array("Red", "Orange", "Green");
```

Combining and cloning Arrays using Spread operator:

Spread operator is a new operator that was introduced as part of JavaScript in 2015. It consists of triple dots (...) which helps in spreading out the elements of an array to a new variable.

When the spread operator is used in the function call, it expands the iterable object, i.e., array into the list of arguments.

Example:

```
1.let numArr = [10, 5, 20];
2.//spread turns array into the list of arguments
3.console.log(Math.max(...numArr)); // 20
```

Arrays can be merged using the spread syntax.

Example:

```
1.let arr1 = [3, 5, 1];
2.let arr2 = [8, 2, 6];
3.let newArr = [...arr1, ...arr2];
4.connsole.log(newArr); // [3,5,1,8,2,6]
```

Arrays can be combined with normal values.

Example:

```
let arr1 = [3, 5, 1];
let arr2 = [8, 2, 6];
let newArr = [0, ...arr1, 4, ...arr2];
console.log(newArr); // [0,3,5,1,4,8,2,6]
```

You can also use the spread operator to create a copy of an array.

Example:

```
1. let arr1 = [3, 5, 1];
2. let arrCopy = [...arr1];
3. arrCopy.push(4);
4. console.log(arrCopy);
5. //arrCopy becomes [3,5,1,4] and arr1 remains unaffected
```

Destructuring Arrays:

JavaScript introduced the destructuring assignment syntax that makes it possible to unpack values from arrays or objects into distinct variables. So, how does this syntax help to unpack values from an array.

Example:

```
1.// [RN1] we have an array with the employee name and id
2.let empArr = ["Shaan", 104567];
3.
4.// destructuring assignment
5.// with empName = empArr[0]
6.// and empId = empArr[1]
7.
8.let [empName, empId] = empArr;
9.console.log(empName); // Shaan
10.console.log(empId);  // 104567
```

Destructuring assignment syntax is just a shorter way to write:

```
1.let empName = empArr[0];
2.let empId = empArr[1];
```

You can also ignore elements of the array using an extra comma:

Example:

```
1.let [empName, , location] = ["Shaan", 104567,
   "Bangalore"];
2.
3.//Here second element of array is skipped and third element
   is assigned to location variable
4.
5.console.log(empName); // Shaan
6.console.log(location);  // Bangalore
```

Rest operator can also be used with destructuring assignment syntax.

Example:

```
1.let [empName, ...rest] = ["Shaan", 104567, "Bangalore"];
2.console.log(empName); // Shaan
3.console.log(rest);  // [104567,"Bangalore"]
```

Here, the value of the rest variable is the array of remaining elements and the rest parameter always goes last in the destructuring assignment.

Accessing Arrays

Array elements can be accessed using indexes. The first element of an array is at index 0 and the last element is at the index equal to the number of array elements – 1. Using an invalid index value returns undefined.

Example:

```
1. let arr = ["first", "second", "third"];
2. console.log(arr[0]); //first
3. console.log(arr[1]); //second
4. console.log(arr[3]); //undefined
```

Loop over an array

You can loop over the array elements using indexes.

Example:

```
1. let colors = ["Red", "Orange", "Green"];
2. for (let i = 0; i < colors.length; i++)
3.   {
4.       console.log(colors[i]);
5. }
6. //Red
7. //Orange
8. //Green
```

Array Methods:

JavaScript arrays consist of several useful methods and properties to modify or access the user-defined array declarations.

Below is the table with property of JavaScript array:

Property	Description	Example

length	It is a read-only property. It returns the length of an array, i.e., number of elements in an array	let myArray = ["Windows", "iOS", "Android"];
		console.log("Length = " + myArray.length);
		//Length = 3

Below is the table with methods to add/remove array elements:

Methods	Description	Example

push()	Adds new element to the end of an array and return the new length of the array	let myArray = ["Android", "iOS", "Windows"];
		myArray.push("Linux");
		console.log(myArray);
		// ["Android","iOS","Windows","Linux"]

pop()	Removes the last element of an array and returns that element	let myArray = ["Android", "iOS", "Windows"];

		console.log(myArray.pop()); // Windows
		
		console.log(myArray); // ["Android", "iOS"]

shift()	Removes the first element of an array and returns that element.	let myArray = ["Android", "iOS", "Windows"];
		
		console.log(myArray.shift()); // Android
		
		console.log(myArray); // ["iOS", "Windows"]

unshift()	Adds new element to the beginning of an array and returns the new length.	let myArray = ["Android", "iOS", "Windows"];
		
		myArray.unshift("Linux");
		
		console.log(myArray);
		
		// ["Linux","Android","iOS","Windows"]

splice()	Changes the content of an array by inserting, removing, and replacing elements. Returns the array of removed elements.
	
	Syntax:
	
	array.splice(index,deleteCount,items);
	
	index = index for new item
	
	deleteCount = number of items to be removed, starting from index next to index of new item
	
	items = items to be added
		let myArray = ["Android", "iOS", "Windows"];
		
		//inserts at index 1:
		
		myArray.splice(1, 0, "Linux");
		
		console.log(myArray);
		
		// ["Android","Linux", "iOS", "Windows"]

slice()	Returns a new array object copying to it all items from start to end (exclusive) where start and end represents the index of items in an array. The original array remains unaffected.
	
	Syntax:
	
	array.slice(startend)
		let myArray=["Android","iOS","Windows"];
		
		console.log(myArray.slice(1,3));
		
		// ["iOS", "Windows"]

concat()	Joins two or more arrays and returns joined array.	let myArray1 = ["Android","iOS"];
		
		let myArray2 = ["Samsung", "Apple"];
		
		console.log(myArray1.concat(myArray2));
		
		// ["Android", "iOS", "Samsung", "Apple"]

[illegible]	[illegible] Syntax: [illegible] [illegible] [illegible] [illegible]	[illegible] console.log(myArray.[illegible]); [illegible] // [illegible] // [illegible] // [illegible] // [illegible]

For more for array methods:

Introduction to Asynchronous Programming:

Synchronous JavaScript: As the name suggests synchronous means to be in a sequence, i.e. every statement of the code gets executed one by one. So, basically a statement has to wait for the earlier statement to get executed.
Let us understand this with the help of an example.

Example:

```
document.write("Hi"); // First
document.write("<br>")

document.write("Mayukh") // Second
document.write("<br>")

document.write("How are you"); // Third
```

Output:

```
Hi
Mayukh
How are you
```

In the above code snippet, the first line of the code **Hi** will be logged first then the second line **Mayukh** will be logged and then after its completion, the third line would be logged **How are you**.
So as we can see the codes work in a sequence. Every line of code waits for its previous one to get executed first and then it gets executed.

Asynchronous JavaScript: Asynchronous code allows the program to be executed immediately, where the synchronous code will block further execution of the remaining code until it finishes the current one. This may not look like a big problem but when you see it in a bigger picture you realize that it may lead to delaying the User Interface.

Let us see the example how Asynchronous JavaScript runs.

```
document.write("Hi")
document.write("<br>");

setTimeout(() => {
    document.write("Let us see what happens")
}, 2000);

document.write("End")
document.write("<br>")
document.write("<br>");
```

```
Hi
End
Let us see what happens
```

So, what the code does is first it logs **Hi**, then rather than executing the **setTimeout** function it logs **End** and then it runs the **setTimeout** function.

At first, as usual, the **log** statement get logged in. As we use browsers to run JavaScript, there are the web APIs that handle these things for users. So, what JavaScript does is, it passes the **setTimeout** function to such web API and then we keep on running our code in usual. So it does not block the rest of the code from executing and after all the code its execution, it gets pushed to the call stack and then finally gets executed. This is what happens in asynchronous JavaScript.

Some of the real-time situations where you may need to use the JavaScript Asynchronous code of execution while implementing business logic are:

- To make an HTTP request call.
- To perform any input/output operations.
- To deal with client and server communications.

These executions in JavaScript can also be achieved through many techniques.

Some of the techniques are:

- Callbacks
- Promises
- Async/Await

Call function:

- A callback function is a function that is passed as an argument to another function. Callbacks make sure that a certain function does not execute until another function has already finished execution.
- Callbacks are handy in case if there is a requirement to inform the executing function on what next when the asynchronous task completes. Here the problem is there are bunch of asynchronous tasks, which expect you to define one callback within another callback and so on. This leads to callback hell.
- **Callback hell**, which is also called a Pyramid of Doom, consists of more than one nested callback, which makes code hard to read and debug. As calls become more nested, the code becomes deeper and increasingly more difficult to manage, especially if there are more loops, conditional statements, and so on in the code.

Example:

```
myFun1(function () {

  myFun2(function () {

    myFun3(function () {

      myFun4(function () {

        ...

      });

    });

  });

});
```

In the above example, it is noticed that the "pyramid" of nested calls grows to the right with every asynchronous action. It leads to callback hell. So, this way of coding is not very good practice. To overcome the disadvantage of callbacks, the concept of Promises was introduced.

Promises:

Promises are used to handle asynchronous operations in JavaScript. They are easy to manage when dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable code. Prior to promises events and callback functions were used but they had limited functionalities and created unmanageable code.
Multiple callback functions would create callback hell that leads to unmanageable code. Also it is not easy for any user to handle multiple callbacks at the same time. Events were not good at handling asynchronous operations.

Promises are the ideal choice for handling asynchronous operations in the simplest manner. They can handle multiple asynchronous operations easily and provide better error handling than callbacks and events. In other words also, we may say that, promises are the ideal choice for handling multiple callbacks at the same time, thus avoiding the undesired callback hell situation. Promises do provide a better chance to a user to read the code in a more effective and efficient manner especially if that particular code is used for implementing multiple asynchronous operations.

- **Benefits of Promises**
  1. Improves Code Readability
  2. Better handling of asynchronous operations
  3. Better flow of control definition in asynchronous logic
  4. Better Error Handling
- **A Promise has four states:**
  1. **fulfilled**: Action related to the promise succeeded
  2. **rejected**: Action related to the promise failed
  3. **pending**: Promise is still pending i.e. not fulfilled or rejected yet
  4. **settled**: Promise has fulfilled or rejected
- **A promise can be created using Promise constructor.**
  Syntax

```
var promise = new Promise(function(resolve, reject){
  //do something
        });
```

Example:

```
var promise = new Promise(function(resolve, reject) {
const x = "geeksforgeeks";
const y = "geeksforgeeks"
if(x === y) {
      resolve();

}
else { reject(); } });
```

```
promise.
        then(function () {
                console.log("Success, You are a GREAT");
        }).
        catch(function () {
                console.log("Some error has occurred");
        });
```

Output:
Success, You are a GREAT

3.Asyic and Await:

async/await" was introduced to implement asynchronous code with promises that resemble synchronous code. "async/await" is simple, easy, readable and understandable than the promises.

Async/Await vs Promises

	Async/Await	Promises
Scope	The entire wrapper function is asynchronous.	Only the promise chain itself is asynchronous.
Logic	• Synchronous work needs to be moved out of the callback. • Multiple promises can be handled with single variables.	• Synchronous work can be handled in the same callback. • Multiple promises use Promise.all().
Error Handling	You can use try, catch and finally	You can use then, catch and finally.

Example 1: In this example, we will see the basic use of async in Javascript

```
<script>
const getData=async()=>{
var data="Hello World";
return data;
}
getData().then(data => console.log(data));
</script>
```

Output: hello world

Await: Await function is used to wait for the promise. It could be used within the async block only. It makes the code wait until the promise returns a result. It only makes the async block wait.

Example 2: This example shows the basic use of the await keyword in Javascript

```
<script>
        const getData = async() => {
                var y = await "Hello World";
                console.log(y);
        }

        console.log(1);
        getData();
        console.log(2);
</script>
```

Output:

```
1
2
Hello World
```

Notice that the console prints 2 before the "**Hello World**". This is due to the usage of the await keyword.

Executing Network Requests using Fetch API:

JavaScript plays an important role in communication with the server. This can be achieved by sending a request to the server and obtaining the information sent by the server. For example:

1. Submit an order,
2. Load user information,
3. Receive latest information updates from the server

All the above works without reloading the current page!

There are many ways to send a request and get a response from the server. The fetch() is a modern and versatile method available in JavaScript.

Fetch provides a generic definition of Request and Response objects. The fetch() method takes one mandatory argument, the path to the resource you want to fetch. It returns a Promise that resolves to Response if the fetch operation is successful or throws an error that can be caught in case the fetch fails. You can also optionally pass in an init options object as the second argument.

Syntax:

```
let response = fetch(resource, [init]).then(...)
```

- resource – The URL to be accessed.
- options – optional parameters: method, headers, etc.

Without options, this is a simple/default GET request, which downloads the contents from the URL. The fetch() returns a promise, which needs to be resolved to obtain the response from the server or for handling the error.

Getting a response from a fetch() is a two-step process.

1. The promise object returned by fetch() needs to be resolved to an object after the server sends a response.

- Here, HTTP status needs to be checked to see it is successful or not.
- The promise will be rejected if the fetch is unable to make a successful HTTP-request to the server e.g. may be due to network issues, or if the URL mentioned in fetch does not exist.
- HTTP-status can be seen in response properties easily by doing console.log
  - status – HTTP status code returned from a response, e.g., 200.
  - ok – Boolean, true if the HTTP status code returned from a response, is 200-299.

2. Get the response body using additional methods.

- Response object can be converted into various formats by using multiple methods to access the body/data from response object:
  - response.text() –read body/data from response object as a text.
  - response.json() – parse body/data from response object as JSON.
  - response.formData() – return body/data from response object as FormData.
  - response.blob() – return body/data from response object as Blob (binary data with its type)

Introduction to Modular Programming:

Modules are one of the most important features of any programming language.

In 2015 modules were introduced in JavaScript officially and they are considered to be first-class citizens while coding the application.

Modules help in state and global namespace isolation and enable reusability and better maintainability.

We need modules in order to effectively reuse, maintain, separate, and encapsulate internal behavior from external behavior.

Each module is a JavaScript file.

Modules are always by default in strict mode code. That is the scope of the members (functions, variables, etc.) which reside inside a module is always local.

The functions or variables defined in a module are not visible outside unless they are explicitly exported.

The developer can create a module and export only those values which are required to be accessed by other parts of the application.

Modules are declarative in nature.

- The keyword "export" is used to export any variable/method/object from a module.
- The keyword "import" is used to consume the exported variables in a different module.

Creating Modules:

The export keyword is used to export some selected entities such as functions, objects, classes, or primitive values from the module so that they can be used by other modules using import statements.

There are two types of exports:

- Named Exports (More exports per module)
- Default Exports (One export per module)

Named exports are recognized by their names. You can include any number of named exports in a module.

There are two ways to export entities from a module.

1. Export individual features

Syntax:

```
1.export let name1, name2, ..., nameN; // Also var, const
2.export let name1 = ..., name2 = ..., ..., nameN;
3.export function functionName(){...}
4.export class ClassName {...}
```

Example:

```
1.export let var1,var2;
2.export function myFunction(){ ... };
```

2. Export List

Syntax:

```
1.export { name1, name2, ..., nameN };
```

Example:

```
1.export { myFunction, var1, var2 };
```

The most common and highly used entity is exported as default. You can use only one default export in a single file.

Syntax:

```
1.export default entityName;
```

where entities may be any of the JavaScript entities like classes, functions, variables, etc.

Example:

```
1.export default function () { ... }
```

Consuming Modules:

If you want to utilize an exported member of a module, use the import keyword. You can use many numbers of import statements.

Syntax:

```
1.//import multiple exports from module
2.import {entity1, entity 2... entity N} from modulename;
3.
4.//import an entire module's contents
5.import * as variablename from modulename;
6.
7.//import an export with more convenient alias
8.import {oldentityname as newentityname } from modulename;
9.
```

Example:

```
1.import {var1,var2} from './mymodule.js';
2.import * as mymodule from './mymodule.js';
3.import {myfunction as foo} from './mymodule.js';
```

You can import a default export with any name.

Syntax:

```
1.import variablename from modulename;
```

Example:

```
1.import mydefault from './mymodule.js';
```