# T3 SKILLS CENTER

**Foundation Day**
**30-09-2023**

# T3 कौशल केंद्र

# TWKSAA JAVASCRIPT BOOK

**Er. Rajesh Prasad**

- आप एक सागर हो बहते नदी का जल नहीं आप एक बदलाव हो भटकाव की कोई राह नहीं
- उस रास्ते पर चलो जिस रास्ते पर भीड़ कम हो (हर काम हो कुछ अलग)
- देश की मिट्टी से करो आप इतना प्यार जहाँ जाओ वहाँ मिले खूब इज्जत और सम्मान
- छह दिन कीजिए अपना काम एक दिन कीजिए त्वक्सा को दान
- त्वक्सा एक चिंगारी हैं हर जगह जलना हम सब की जिमेवारी हैं
- **Motive: - New (RID PMS & TLR)**

"त्वक्सा जावा स्क्रीप्ट के इस पुस्तक में आप कोर जावा स्क्रीप्ट के संबंध में सभी बुनियादी अवधारणाएँ सीखेंगे। मुझे आशा है कि इस पुस्तक को पढ़ने के बाद आपके ज्ञान में वृद्धि होगी और आपको कंप्यूटर विज्ञान के बारे में और अधिक जानने में रुचि होगी"

"In this TWKSAA JavaScript book you will learn all basic concept regarding core python. I hope after reading this book your knowledge will be improve and you will get more interest to know more thing about computer Science".

"Skill कौशल एक व्यक्ति के पास उनके ज्ञान, अनुभव, तत्वशास्त्रीय योग्यता, और प्रैक्टिकल अभियांत्रिकी के साथ संचित नौकरी, व्यापार, या अन्य चुनौतीपूर्ण परिस्थितियों में सक्रिय रूप से काम करने की क्षमता को कहते हैं। यह व्यक्ति के द्वारा सीखी जाने वाली कौशलों की प्रतिभा, क्षमता और निपुणता को संक्षेप में व्यक्त करता है"।

## TWKSAA RID MISSION

| (Research) | (Innovation) | (Discovery) |
|---|---|---|
| **अनुसंधान करने के महत्वपूर्ण कारण:** | **नवीनीकरण करने के महत्वपूर्ण कारण:** | **खोज करने के महत्वपूर्ण कारण:** |
| 1. नई ज्ञान की प्राप्ति | 1. प्रगति के लिए | 1. नए ज्ञान की प्राप्ति |
| 2. समस्याओं का समाधान | 2. परिवर्तन के लिए | 2. ज्ञान के विकास में योगदान |
| 3. तकनीकी और व्यापार में उन्नति | 3. उत्पादन में सुधार | 3. अविष्कारों की खोज |
| 4. विकास को बढ़ावा देना | 4. प्रतिस्पर्धा में अग्रणी होने के लिए | 4. समस्याओं का समाधान |
| 5. सामाजिक प्रगति | 5. समाज को लाभ | 5. समाज के उन्नति का माध्यम |
| 6. देश विज्ञान और प्रौद्योगिकी का विकास | 6. देश विज्ञान और प्रौद्योगिकी के विकास। | 6. देश विज्ञान और तकनीक के विकास |

**"T3 Skills Center is a Learning Earning and Development Based Skill Center."**

**T3 कौशल केंद्र एक सीखने कमाई और विकास आधारित कौशल केंद्र है।**

# T3 SKILLS CENTER

# JAVASCRIPT

- JavaScript is a dynamic programming language.
- It is lightweight and most commonly use as a part of web pages.
- It is interpreted programming language
- It is platform independent programming language.
- It is object-oriented programming language.
- JavaScript is used to modify the HTML content
- In HTML, JavaScript code is inserted between <script>……..</script> tags.
- You can place any number of scripts in an HTML page.

## ❖ Features of JavaScript:

- **High-level Language:** it is a high-level programming language,means Human readable.
- **Interpreted Language:** JavaScript is typically executed by a web browser's JavaScript engine or a server-side runtime environment like Node.js. It doesn't require compilation before execution, making it a scripting language.
- **Dynamic Typing:** it is dynamically typed, allowing variables to change types at runtime.
- **Weak Typing:** JavaScript is also weakly typed, which means it performs type coercion automatically when operations involve different data types.
- **Client-Side Scripting:** JavaScript is primarily used for client-side scripting in web development. It runs directly in web browsers and can manipulate the Document Object Model (DOM) to change the content and behaviour of web pages dynamically.
- **Cross-Platform:** JavaScript is supported by all major web browsers, making it a cross-platform language.
- **Object-Oriented:** JavaScript is an object-oriented language, supporting the creation of objects and classes.
- **First-Class Functions:** JavaScript treats functions as first-class citizens, meaning they can be assigned to variables, passed as arguments to other functions, and returned from functions.
- **Closures:** JavaScript supports closures, which allow inner functions to access variables from their outer containing functions even after the outer function has finished executing. Closures are crucial for maintaining state and data encapsulation.
- **Asynchronous Programming:** JavaScript excels in handling asynchronous operations, such as AJAX requests, timers, and event-driven programming. Call-backs, Promises, and the async/await syntax make it easier to manage asynchronous code.
- **Modularity:** JavaScript allows you to organize code into modules, improving code maintainability and reusability. The ES6 module system, introduced in ECMAScript 2015, provides a standardized way to define and import/export modules.
- **Extensible:** JavaScript can be extended through libraries, frameworks, and external APIs. A rich ecosystem of third-party libraries and tools exists to enhance JavaScript's capabilities.
- **Standardized:** JavaScript is standardized through the ECMAScript specification, which defines the language's core features.
- **Security:** it has built-in security features to protect against malicious code execution.
- **Community and Ecosystem:** JavaScript has a vast and active developer community, contributing to its continuous growth and innovation.

- **Scalability:** With the advent of server-side JavaScript via Node.js, JavaScript can be used for both front-end and back-end development, making it a suitable choice for building scalable, full-stack applications.

## ❖ History of JavaScript:

- Sir Tim Berners Lee introduced web in early 1990's
- Sir Tim Berners introduced a language called HTML.
- In 1993, Mosaic, the first popular web browser, came into existence. In the year 1994, Netscape was founded by Marc Andreessen. He realized that the web needed to become more dynamic. Thus, a 'glue language' was believed to be provided to HTML to make web designing easy for designers and part-time programmers. Consequently, in 1995, the company recruited Brendan Eich intending to implement and embed Scheme programming language to the browser. But, before Brendan could start, the company merged with Sun Microsystems for adding Java into its Navigator so that it could compete with Microsoft over the web technologies and platforms. Now, two languages were there: Java and the scripting language. Further, Netscape decided to give a similar name to the scripting language as Java's. It led to 'JavaScript'. Finally, in May 1995, Marc Andreessen coined the first code of JavaScript named 'Mocha'. Later, the marketing team replaced the name with 'Live Script'. But, due to trademark reasons and certain other reasons, in December 1995, the language was finally renamed to 'JavaScript'. From then, JavaScript came into existence.

## ❖ Application of JavaScript:

- **Web Development:** JavaScript is primarily used for creating dynamic and interactive web applications.
- **Front-End Web Development:** JavaScript is a core technology for front-end web development. Popular libraries and frameworks like React, Angular, and Vue.js are built using JavaScript to simplify the process of building complex user interfaces.
- **Single-Page Applications (SPAs):** SPAs load a single HTML page and dynamically update content as the user interacts with the application. JavaScript frameworks like Angular, React, and Vue.js are commonly used for building SPAs.
- **Mobile App Development:** With tools like React Native and NativeScript, developers can use JavaScript to build mobile applications for iOS and Android platforms.
- **Backend Development:** While JavaScript is mainly associated with front-end development, it can also be used for backend development. Node.js, a runtime environment, allows developers to run JavaScript on the server side, making it possible to build full-stack web applications using a single language.
- **Game Development:** JavaScript, combined with HTML5 and canvas, can be used to create simple web-based games. Libraries like Phaser and Three.js
- **Web APIs:** JavaScript can interact with various web APIs, enabling developers to fetch data from external sources, such as social media platforms, weather services, and databases, to display and manipulate that data on web pages.
- **Web Animation:** JavaScript can be used to create animations and special effects on web pages. The HTML5 <canvas> element and CSS transitions/animations are often used in combination with JavaScript for this purpose.
- **Browser Extensions:** JavaScript can be used to create browser extensions that add new features or functionality to web browsers like Chrome and Firefox.
- **Data Visualization:** JavaScript libraries such as D3.js and Chart.js are used to create interactive data visualizations and charts on web pages,

- **Real-Time Applications:** JavaScript is used to develop real-time applications like chat applications, online gaming, and collaborative tools where data needs to be updated and synchronized in real-time.
- **Cross-Platform Desktop Applications:** Frameworks like Electron and NW.js enable developers to build cross-platform desktop applications using JavaScript, HTML, and CSS.
- **Internet of Things (IoT):** JavaScript can be used in IoT development, especially for building web-based dashboards and interfaces to control and monitor IoT devices.
- **Serverless Computing:** JavaScript can be used in serverless computing platforms like AWS Lambda, Azure Functions, and Google Cloud Functions to create serverless applications and microservices.
- **Automation:** JavaScript can be used in conjunction with headless browsers like Puppeteer to automate tasks such as web scraping, testing, and filling out forms.
- **Machine Learning and AI:** JavaScript libraries like TensorFlow.js and Brain.js provide tools for running machine learning and artificial intelligence models in the browser.

## ❖ JavaScript Comment:

- The JavaScript comments are meaningful way to deliver message. It is used to add information about the code, warnings or suggestions

### ➢ Types of JavaScript Comments

- There are two types of comments in JavaScript.
    1. Single-line Comment
    2. Multi-line Comment

### ➢ JavaScript Single line Comment

- It is represented by double forward slashes (//). It can be used before and after the statement.

**Example:**

```
<script>
var a=10;
var b=20;
var c=a+b;//It adds values of a and b variable
console.log(c)
document.write(c);//prints sum of 10 and 20
</script>
```

### ➢ JavaScript Multi line Comment:

- It can be used to add single as well as multi line comments. So, it is more convenient.

**Example:**

```
/* It is multi line comment.
It will not be displayed */
```

**Example:**

```
/*console.log("welcome to twksaa skills center")
let a=20
let b=10
let c=a+b
console.log(c) */
var d=30
console.log("value of d=",d)
```

**Output:**

```
PS C:\Users\hp\OneDrive\Desktop\javascript> node ./print.js   value of d= 30
```

## ❖ JavaScript Variable:

- A JavaScript variable is simply a name of storage location.
- There are two types of variables in JavaScript:
  1. local variable and
  2. global variable.

➢ **Rules for declaring a JavaScript variable (also known as identifiers).**

- Name must start with a letter (a to z or A to Z), underscore (_ ), or dollar( $ ) sign.
- After first letter we can use digits (0 to 9), for example value1.
- JavaScript variables are case sensitive, for example a and A are different variables.

➢ **JavaScript local variable:**

- A JavaScript local variable is declared inside block or function. It is accessible within the function or block only.

**Example:**

```
<script>
function abc(){
var x=10;//local variable
}
</script>
```

**Or**

```
<script>
If(10<15){
var y=20;//JavaScript local variable
}
</script>
```

## ❖ JavaScript Identifiers:

- All JavaScript variables must be identified with unique names.
- These unique names are called identifiers.
- Rules of naming variables in js:

1. special keywords shouldn't use example-if,for,let etc.
2. Names must begin with a letter.
3. No space allowed
4. Name can start with underscore or alphabets
5. Names can contain letters, digits, underscores, and dollar signs.

## ❖ Naming conventions:

1.camelCase- ex- addOfTwoNumbers
2.PascalCase- ex-AddOfTwoNumbers
3.snake_case- ex-add_of_two_numbers

## ❖ Variable Declaration:

- JavaScript Variables can be declared in 4 ways:
  1. Automatically
  2. Using var
  3. Using let
  4. Using const

**Note:** ; semicolon is optional in javascript

**Example:**

- a, b, and c are undeclared variables.
- They are automatically declared when first used:

```
a=6
b=9
c=a+b
console.log(c)
```

**output:**

```
PS D:\js rev\rajt3> node ./p1.js
15
```

- From the above examples you can guess:
  a store the value 6
  b stores the value 9
  c stores the value 15

## Example using var:

```
var x = 9;
var y = 6;
var z = x + y;
console.log(z)
```

**output:**

```
15
```

**Note:**

- The var keyword was used in all JavaScript code from 1995 to 2015.
- The let and const keywords were added to JavaScript in 2015.
- The var keyword should only be used in code written for older browsers.

## Example using let:

```
let x = 25;
let y = 50;
let z = x + y;
console.log(z)
```

**Output:**

```
PS D:\js rev\rajt3> node ./p1.js
75
```

- in this above output we are getting outside of Browser by using Node.js

**Example for run this t below program through browser**

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Variables</h1>
<p>In this example, x, y, and z are variables.</p>
<p id="demo"></p>
<script>
let x = 25;
let y = 50;
let z = x + y;
document.getElementById("demo").innerHTML =
```

```
"The value of z is: " + z;
</script>
</body>
</html>
```

**output:**

```
JavaScript Variables
In this example, x, y, and z are variables.
The value of z is: 75
```

## Example using const:

```
const x = 25;
const y = 50;
const  z = x + y;
console.log(z)
```

**output:**

```
PS D:\js rev\rajt3> node ./p1.js
75
```

## Mixed Example:

```
const Num1= 15
const Num2 = 20
let Sum = Num1 + Num2
console.log("Sum of Num1 and Num2=",Sum)
```

**output:**

```
Sum of Num1 and Num2= 35
```

**Note:**

- The two variables Num1 and Num2 are declared with the const keyword.
- These are constant values and cannot be changed.
- The variable Sum is declared with the let keyword.
- The value Sum can be changed.

## When to Use var, let, or const?

1. Always declare variables
2. Always use const if the value should not be changed
3. Always use const if the type should not be changed (Arrays and Objects)
4. Only use let if you can't use const
5. Only use var if you MUST support old browsers.

## Difference between var const and let in JavaScript:

### ❖ var:

- Function-scoped: Variables declared with var are function-scoped, meaning they are only accessible within the function in which they are declared. If declared outside of any function, they become globally scoped.
- Hoisting: Variables declared with var are hoisted to the top of their function or global scope. This means you can use a var variable before it's declared in code, but it will have an initial value of undefined.
- Reassignable: You can reassign values to a var variable.

## ❖ let:

- Block-scoped: Variables declared with let are block-scoped, meaning they are only accessible within the block (inside curly braces) in which they are declared, or within sub-blocks.
- Hoisting: Like var, let variables are hoisted, but they are not initialized to undefined. If you try to access a let variable before its declaration in code, you'll get a ReferenceError.
- Reassignable: You can reassign values to a let variable.

## ❖ const:

- Block-scoped: Like let, const variables are block-scoped.
- Hoisting: const variables are hoisted, but like let, they are also not initialized to undefined. Accessing a const variable before its declaration in code will result in a ReferenceError.
- Immutability: Variables declared with const cannot be reassigned once they are given a value. However, the value itself may be mutable if it's an object or an array. This means you can change the properties or elements of a const object or array.

**Example:**

```
var is global scope
let & const are block scope
In JS, we can use of pair of empty curly brackets to create an empty block.
{
var c=10
let d=20
console.log(c,d)
}
console.log(c) //10
console.log(d)//It will throw error as d can be accessed inside local block here
const e=20//const variables can't be declared separately but initialise also
```

**Example:**

```
let a="10"
let b=20
console.log(a+b+++a)//a+b++ +a=102010
console.log(a,b)//a="10" b=21
console.log(a+b+++b)//a+b+ ++b=102122
```

## ❖ Block Scope:

- Before ES6 (2015), JavaScript had Global Scope and Function Scope.
- ES6 introduced two important new JavaScript keywords: let and const.
- These two keywords provide Block Scope in JavaScript.
- Variables declared inside a { } block cannot be accessed from outside the block:

## Note:

- let and const have block scope.
- let and const can not be redeclared.
- let and const must be declared before use.
- let and const does not bind to this.
- let and const are not hoisted.
- var does not have to be declared.
- var is hoisted.
- var binds to this.

## let:

- let Cannot be Redeclared:
- Variables defined with let can not be redeclared.

**Example:**

```
let a=15
console.log(a)
let a=20 //let can not be redeclared
console.log(a)
```

**But if you will be declaring variable with let in same name inside and outside of the {} block**

**Example:**

```
let a = 15// Here x is 15
{
let a = 20// Here x is 20
console.log("value of a=",a)
}
console.log("value of a=",a) // Here x is 15
```

**output:**

```
value of a= 20
value of a= 15
```

- let can be declare without assign value:

**Example:**

```
let fd // let can declear variable without assign value also
fd=50
console.log(fd)
```

**output:**

```
50
```

## Var:

- var can be Redeclared:
- Variables defined with var can be redeclared allowed anywhere in a program.

**Example-1:**

```
var a=15
console.log("value of a=",a)
var a=20 //var can be redeclared
console.log("value of a=",a)
```

**output:**

```
value of a= 15
value of a= 20
Example-2:
var a = 15// Here x is 15
{
var a = 20// Here x is 20
console.log("value of a=",a)
}
console.log("value of a=",a) // Here x is 15
```

**output:**

```
value of a= 20
```

value of a= 20

- var can be declare without assign value:

**Example:**

```
var fd // var can declear variable without assign value also
fd=25
console.log(fd)
```

**output:**

25

## const:

- The const keyword was introduced in ES6 (2015)
- Variables defined with const cannot be Redeclared
- Variables defined with const cannot be Reassigned
- Variables defined with const have Block Scope
- Const can not redeclared:

**Example-1**

```
const a = 15// Here x is 15
console.log("value of a=",a)
const a = 20// Here x is 20 can not declear in the case of const
console.log("value of a=",a) // Here x is 15
```

**Example-2:**

- But const can be redeclared with same variable name inside and outside of {} block

```
const a = 15// Here x is 15
{
const a = 20// Here x is 20
console.log("value of a=",a)
}
console.log("value of a=",a) // Here x is 15
```

**output:**

```
value of a= 20
value of a= 20
```

- const Cannot be Reassigned:
  - A const variable cannot be reassigned:

**Example:**

```
const fd= 30
fd = 9     // This will give an error
fd = fd + 23   // This will also give an error
console.log(fd)
console.log(fd)
console.log(fd)
```

- const Must be Assigned:
- JavaScript const variables must be assigned a value when they are declared:

**Example-1:**

```
const fd= 30
console.log(fd)
```

**output:**

30

**Example-2:**

```
const fd // cant declear variable without assign value in case of const
fd=30  // error
console.log(fd)
```

### ❖ When to use JavaScript const?

- Always declare a variable with const when you know that the value should not be changed.
- Use const when you declare:
- A new Array
- A new Object
- A new Function
- A new RegExp

# JavaScript Operators:

- JavaScript operators are symbols that are used to perform operations on operands.
- Types of JavaScript Operators
  - There are different types of JavaScript operators:
    1. Arithmetic Operators
    2. Assignment Operators
    3. Comparison Operators
    4. Logical Operators
    5. Bitwise Operators
    6. Type Operators
    7. Ternary Operators

### ❖ Arithmetic Operators:

- Arithmetic Operators are used to perform arithmetic operation:

**Example:**

| Operator | Description | Example |
|---|---|---|
| + | Addition | 15+20 = 35 |
| - | Subtraction | 50-25 = 25 |
| * | Multiplication | 3*6 = 18 |
| / | Division | 40/8 = 5 |
| % | Modulus (Remainder) | 20%10 = 0 |
| ++ | Increment | var a=20; a++; Now a = 21 |
| -- | Decrement | var a=20; a--; Now a = 19 |

### ❖ Assignment Operators:

- Assignment operators assign values to JavaScript variables.

**Example:**

| Operator | Description | Example |
|---|---|---|
| = | Assign | 10+10 = 20 |
| += | Add and assign | var a=10; a+=20; Now a = 30 |
| -= | Subtract and assign | var a=20; a-=10; Now a = 10 |
| *= | Multiply and assign | var a=10; a*=20; Now a = 200 |
| /= | Divide and assign | var a=10; a/=2; Now a = 5 |
| %= | Modulus and assign | var a=10; a%=2; Now a = 0 |

## ❖ *Comparison Operators:*

| Operator | Description | Example |
|---|---|---|
| == | Is equal to | 10==20 = false |
| === | Identical (equal and of same type) | 10==20 = false |
| != | Not equal to | 10!=20 = true |
| !== | Not Identical | 20!==20 = false |
| > | Greater than | 20>10 = true |
| >= | Greater than or equal to | 20>=10 = true |
| < | Less than | 20<10 = false |
| <= | Less than or equal to | 20<=10 = false |

## ❖ Logical Operators:

- The following operators are known as JavaScript logical operators.

| Operator | Description | Example |
|---|---|---|
| && | Logical AND | (10==20 && 20==33) = false |
| \|\| | Logical OR | (10==20 \|\| 20==33) = false |
| ! | Logical Not | !(10==20) = true |

## ❖ Bitwise Operators:

- bitwise operators perform bitwise operations on operands. bitwise operators are as follows:

| Operator | Description | Decimal | Example | Same as | Result |
|---|---|---|---|---|---|
| & | AND | 5 & 1 | 0101 & 0001 | 0001 | 1 |
| \| | OR | 5 \| 1 | 0101 \| 0001 | 0101 | 5 |
| ~ | NOT | ~ 5 | ~0101 | 1010 | 10 |
| ^ | XOR | 5 ^ 1 | 0101 ^ 0001 | 0100 | 4 |
| << | left shift | 5 << 1 | 0101 << 1 | 1010 | 10 |
| >> | right shift | 5 >> 1 | 0101 >> 1 | 0010 | 2 |
| >>> | unsigned right shift | 5 >>> 1 | 0101 >>> 1 | 0010 | 2 |

## ❖ Type Operators:

| Operator | Description |
|---|---|
| typeof | Returns the type of a variable |
| instanceof | Returns true if an object is an instance of an object type |

## ❖ Ternary Operators:

- ternary operator, also known as the conditional operator, is a concise way to write conditional statements. It takes three operands and returns a value based on a condition.

**Syntax:**

- condition ? expressionIfTrue : expressionIfFalse
- condition: A boolean expression that is evaluated. If it's true, the expression before the : is executed; otherwise, the expression after the : is executed.
- expressionIfTrue: The value or expression to be returned if the condition is true.
- expressionIfFalse: The value or expression to be returned if the condition is false.

## ❖ Special Operators:

- The following operators are known as JavaScript special operators.

| Operator | Description |
|---|---|
| , | Comma Operator allows multiple expressions to be evaluated as single statement. |
| delete | Delete Operator deletes a property from the object. |
| in | In Operator checks if object has the given property |
| new | creates an instance (object) |
| void | it discards the expression's return value. |
| yield | checks what is returned in a generator by the generator's iterator. |

## ❖ Arithmetic Operators:

**Example:**

```
// let a=20
// let b=30
//DYNAMIC INPUT IN node js
//Install the module prompt-sync
// npm i prompt-sync
const prompt = require("prompt-sync")();
let a = parseInt(prompt("Num 1- "))
let b = parseInt(prompt("Num 2- "))
let sum=a+b
let sub=a-b
let Mul=a*b
let Div=a/b
let modulus=a%b
let pre_inc=++a
let post_inc=a++
let pre_dec=--a
let post_dec=a--
console.log("Sum of a and b=",sum)
console.log("Subtraction of a and b=",sub)
console.log("Multiplication of a and b=",Mul)
console.log("Division of a and b=", Div.toFixed(3))
console.log("Modulus or Remainder of a and b=",modulus)
console.log("Pre increment value of a=",pre_inc)
console.log("Post increment value of a=",post_inc)
console.log("Pre decrement value of a=",pre_dec)
console.log("Post decrement value of a=",post_dec)
```

**Output:**

```
PS D:\js rev\rajt3> npm i prompt-sync
added 3 packages in 850ms
PS D:\js rev\rajt3> node ./p1.js
Num 1- 20
Num 2- 10
Sum of a and b= 30
Subtraction of a and b= 10
```

Multiplication of a and b= 200
Division of a and b= 2
Modulus or Remainder of a and b= 0
Pre increment value of a= 21
Post increment value of a= 21
Pre decrement value of a= 21
Post decrement value of a= 21

**Note:**

```
let a=20
console.log(a) //20
console.log(++a) //21
console.log(a++) //21
console.log(a)//22
console.log(--a)//21
console.log(a)//21
console.log(a--)//21
console.log(a)//20
```

## ❖ Assignment Operators:

**Example:**

```
// = is assignment operator. It is used to store value inside a variable
let a=20
let b=30
a+=b//a=a+b
console.log(a)
a-=b//a=a-b
console.log(a)
a*=b//a=a*b
console.log(a)
a/=b//a=a/b
console.log(a)
a%=b// a=a%b
console.log(a,b%a)
b=2
a**=b
console.log(a)
```

Output:

```
PS D:\js rev> node ./AssigOp.js
50
20
600
20
20 10
400
```

## ❖ Compression Operator:

Example:

```
1.Equality(==)  It only checks value not datatype
let a=10
```

```
let b="10"
console.log(a==b);
2. Strict Equality(===) It checks value & datatype both
a=10
b="10"
console.log(a===b);
3. Inequality (!=):  Compares inequality of two operands. True if operands are not equal.
a=10
b="10"
let c="20"
let d=30
console.log(a!=b,c!=d);
4.Strict inequality(!==): Compares the inequality of two operands with type. If both value and
type are equal then the condition is false otherwise true.
a=10
b="10"
c=50
d=50
console.log(a!==b,c!==d);
let x=20
let y=10
console.log(x>y)
console.log(x<y)
console.log(x>=y)
console.log(x<=y)
```

**output:**

```
true
false
false true
true false
true
false
true
false
```

## ❖ Logical Operators:

**Example:**

```
//LOGICAL OPERATORS-
// 1. AND(&&)
// 2. OR(||)
// 3. NOT(!)
let a=20
let b=30
let c=10
console.log(a>b && a>c)//false
console.log(a>b || a>c)//true
console.log(!(a>b && a>c))//(a>b && a>c) is false but adding NOT(!) makes the o/p true
console.log(!(a>b || a>c))//(a>b || a>c) is true but adding NOT(!) makes the o/p false
```

**Output:**

false
true
true
false

❖ **Bitwise Operators:**

**Example:**

```
let a=10
let b=12
console.log(a&b)
console.log(a|b)
console.log(a^b)
console.log(~b,~a)
console.log(a<<2)
console.log(b>>2,a>>2)
```

**Output:**

8
14
6
-13 -11
40
3 2

❖ **Type Operators:**

**Example:**

```
// Define variables with different data types
const number = 25;
const string = "TWKSAA SKILLS CENTER";
const boolean = true;
const array = [1, 2, 3,4,5,6];
const object = { name: "Sangam", age: 12 };
const func = function () {};
// Use typeof to check data types
console.log(typeof number);  // "number"
console.log(typeof string);  // "string"
console.log(typeof boolean); // "boolean"
console.log(typeof array);   // "object" (Arrays are objects in JavaScript)
console.log(typeof object);  // "object"
console.log(typeof func);    // "function"
console.log(typeof undefined);  // "undefined"
console.log(typeof null);      // "object" (null is a special case)
```

**Output:**

number
string
boolean
object
object

function
undefined
object

## ❖ instanceof operator in JavaScript:

**Example:**

```
// Define constructor functions for custom objects
function Animal(name) {
  this.name = name;
}
function Dog(name, breed) {
  Animal.call(this, name); // Call the Animal constructor to set the name property
  this.breed = breed;
}
// Create instances of objects
const dog1 = new Dog("Buddy", "Golden Retriever");
const dog2 = new Dog("Rex", "German Shepherd");
const animal = new Animal("Unknown");
// Use the instanceof operator to check object types
console.log(dog1 instanceof Dog);    // true, dog1 is an instance of Dog
console.log(dog1 instanceof Animal);  // true, dog1 is also an instance of Animal (due to prototype chain)
console.log(animal instanceof Dog);   // false, animal is not an instance of Dog
console.log(animal instanceof Animal); // true, animal is an instance of Animal
```

**Output:**

```
true
false
false
true
```

## ❖ Ternary Operators:

- Operator: The “Question mark” or “conditional” operator in JavaScript is a ternary operator that has three operands. It is the simplified operator of if/else.

**Syntax:**

➢ condition ? "statements if condition is true" : "statements if condition is false"

**Example:**

```
let n1=50
let n2=20
// if(n1>n2)
// console.log(n1)
// else
// console.log(n2)
let max= n1>n2 ? n1 : n2
console.log(max)
```

**Output:**

```
50
```

# DATA TYPES

- JavaScript provides different data types to hold different types of values.
- There are two types of data types in JavaScript.
    a. Primitive data type
    b. Non-primitive data type

**Note:** JavaScript is a dynamic type language, means you don't need to specify type of the variable because it is dynamically used by JavaScript engine.

## ❖ primitive data types:

- also known as inbuilt data types:
- There are following types of primitive data types
    **1)** Number
    **2)** String
    **3)** Boolean
    **4)** Undefined
    **5)** Null
    **6)** Symbol (ES6)
    **7)** BigInt (ES11)

## ❖ Number:

- The Number data type is used to represent numeric values, which can be integers or floating-point numbers.
- Numbers are used for mathematical calculations and numerical operations.

**Example:** const age = 30;

## ❖ String:

- The String data type is used to represent sequences of characters enclosed in single or double quotes.
- Strings are widely used for representing text and can include letters, numbers, symbols, and spaces.

**Example:** const name = "Sangam Kumar";

## ❖ Boolean:

- The Boolean data type represents a binary value, which can be either true or false.
- Booleans are primarily used for logical operations and comparisons.

**Example:** const isStudent = true;

## ❖ Undefined:

- The Undefined data type represents a variable that has been declared but has not been assigned a value.
- Variables that are declared but not initialized are automatically assigned the value undefined.
- It is also used as the default return value of functions that do not explicitly return a value.

**Example:** let x; // x is undefined

❖ Null:

- The Null data type represents the intentional absence of any object value or a placeholder for an object that does not exist or is unknown.
- It is often used to indicate that a variable intentionally has no value.

**Example:** const emptyValue = null;

## ❖ Symbol (ES6):

- The Symbol data type, introduced in ECMAScript 6 (ES6), represents a unique and immutable value.
- Symbols are often used as object property keys to prevent unintentional property name collisions.
- They are guaranteed to be unique, even if two symbols have the same description.

**Example:** const uniqueKey = Symbol("unique");

## ❖ BigInt (ES11):

- The BigInt data type, introduced in ECMAScript 11 (ES11), is used to represent arbitrarily large integers that exceed the limits of the Number type.
- BigInts are created by appending n to the end of an integer literal or by using the BigInt() constructor.
- They are useful for working with very large numbers, such as when dealing with cryptographic operations or large integers.

**Example:** const bigIntValue = 689012345678901234567890n;

**Program:**

```
// Number
const age = 15;
// String
const name = "Sangam Kumar";
// Boolean
const isStudent = true;
// Undefined
let a;
// Null
const emptyValue = null;
// Symbol (ES6)
const uniqueKey = Symbol("unique");
// BigInt (ES11)
const bigIntValue = 678901234567890n;
// Check the data types
console.log(`Data type of age: ${typeof age}`);            // "number"
console.log(`Data type of name: ${typeof name}`);          // "string"
console.log(`Data type of isStudent: ${typeof isStudent}`);  // "boolean"
console.log(`Data type of x: ${typeof x}`);                // "undefined"
console.log(`Data type of emptyValue: ${typeof emptyValue}`); // "object" (Note: null is considered an object)
console.log(`Data type of uniqueKey: ${typeof uniqueKey}`);  // "symbol"
console.log(`Data type of bigIntValue: ${typeof bigIntValue}`); // "bigint"
```

**Output:**

```
Data type of age: number
Data type of name: string
Data type of isStudent: boolean
Data type of x: undefined
Data type of emptyValue: object
Data type of uniqueKey: symbol
Data type of bigIntValue: bigint
```

## ❖ non-primitive data types:

- The data types that are derived from primitive data types of the JavaScript language are known as non-primitive data types. It is also known as derived data types or reference data types.
- Non-primitive data types in JavaScript, also known as reference types, are data structures that can hold multiple values and are mutable, including objects, arrays, and functions.

❖ Types of non-primitive data types:

1. **Object:** Objects are complex data types that can store key-value pairs and represent various entities in JavaScript. They include regular objects, arrays, functions, and more.
2. **Array:** Arrays are ordered lists of values, and they can contain elements of various data types. Arrays are used for storing collections of data.
3. **Function:** Functions are reusable blocks of code that can be invoked to perform a specific task or calculation. Functions are also objects in JavaScript.
4. **Date:** The Date object represents dates and times. It is used for working with date and time-related operations.
5. **RegExp:** The RegExp (regular expression) object is used for pattern matching within strings. It allows you to perform powerful text searching and manipulation.
6. **Map:** The Map object is a collection of key-value pairs where keys can be of any data type. It is often used when you need to associate values with specific keys.
7. **Set:** The Set object is a collection of unique values. It is used when you need to store a list of distinct items.
8. **Symbol:** Symbols can also be considered non-primitive, although they are often categorized as primitive data types. They are used for creating unique property keys in objects.

**Program:**

```
// Object
const person = {
  firstName: "Sangam",
  lastName: "Kumar",
  age: 15
}
// Array
const numbers = [15, 20, 25, 50, 53]
// Function
function add(a, b) {
  return a + b
}
// Date
const currentDate = new Date();
// RegExp
const regexPattern = /hello/i;
const sampleString = "TWKSAA SKILLS CENTER";
const isMatch = regexPattern.test(sampleString);
// Map
const fruitMap = new Map();
fruitMap.set("apple", 3);
fruitMap.set("banana", 6);
fruitMap.set("cherry", 9);
```

```
// Set
const uniqueNumbers = new Set([1, 2, 3, 2, 4, 5, 4]);
// Symbol
const uniqueKey = Symbol("unique");
console.log("Object:", person);
console.log("Array:", numbers);
console.log("Function Result:", add(3, 9));
console.log("Current Date:", currentDate);
console.log("RegExp Test Result:", isMatch);
console.log("Map:", fruitMap);
console.log("Set:", uniqueNumbers);
console.log("Symbol:", uniqueKey);
```

**output:**

```
Object: { firstName: 'Sangam', lastName: 'Kumar', age: 15 }
Array: [ 15, 20, 25, 50, 53 ]
Function Result: 12
Current Date: 2023-09-14T14:25:32.365Z
RegExp Test Result: false
Map: Map(3) { 'apple' => 3, 'banana' => 6, 'cherry' => 9 }
Set: Set(5) { 1, 2, 3, 4, 5 }
Symbol: Symbol(unique)
```

**Example:**

Check the data types:

```
// Object
const person = {
  firstName: "Sangam",
  lastName: "Kumar",
  age: 15
};
// Array
const numbers = [1, 2, 3, 4, 5,6];
// Function
function add(a, b) {
  return a + b;
}
// Date
const currentDate = new Date();
// RegExp
const regexPattern = /hello/i;
const sampleString = "TWKSAA SKILLS CENTER";
const isMatch = regexPattern.test(sampleString);
// Map
const fruitMap = new Map();
fruitMap.set("apple", 3);
fruitMap.set("banana", 6);
fruitMap.set("cherry", 9);
// Set
const uniqueNumbers = new Set([1, 2, 3, 2, 4, 5, 4]);
```

```
// Symbol
const uniqueKey = Symbol("unique");
// Check data types
console.log(`Data type of person: ${typeof person}`);           // "object"
console.log(`Data type of numbers: ${typeof numbers}`);         // "object"
console.log(`Data type of add function: ${typeof add}`);        // "function"
console.log(`Data type of currentDate: ${typeof currentDate}`); // "object"
console.log(`Data type of isMatch: ${typeof isMatch}`);         // "boolean"
console.log(`Data type of fruitMap: ${typeof fruitMap}`);       // "object"
console.log(`Data type of uniqueNumbers: ${typeof uniqueNumbers}`); // "object"
console.log(`Data type of uniqueKey: ${typeof uniqueKey}`);     // "symbol"
```

**output:**

```
Data type of person: object
Data type of numbers: object
Data type of add function: function
Data type of currentDate: object
Data type of isMatch: boolean
Data type of fruitMap: object
Data type of uniqueNumbers: object
Data type of uniqueKey: symbol
```

## ❖ Difference b/w primitive & non-primitive data types:

**Example:**

```
var bigNum = 12342222222222222222222222222222n
```

BigInt: This data type can represent numbers greater than 253-1 which helps to perform operations on large numbers. The number is specified by writing 'n' at the end of the value

```
console.log(typeof bigNum)
//PRIMITVE-
let num1=20;
let num2=num1;
console.log(num1,num2)
num1=30
console.log(num1,num2)
//NON-PRIMITVE
let a=[1,2,3]
let b=a
console.log(a,b)
a[1]=55
console.log(a,b)
//Note- In case of primitve data type, the changes wil be reflected in modified variable only
//but
//In case of non-primitve data type,the changes will be reflected in both the copies
/*
Primitve data types hold immutable values
Non-Primitive data types hold mutable values.
*/
// let a=20;
```

```
// a=40;
// let c=[1,2,3]
// c[0]=20
```

**output:**

```
bigint
20 20
30 20
[ 1, 2, 3 ] [ 1, 2, 3 ]
[ 1, 55, 3 ] [ 1, 55, 3 ]
```

# CONDITIONAL STATEMENTS

- There are three main types of conditional statements:
    1. if statements,
    2. if-else statements,
    3. else-if ladders.

## 1. if Statement:

- The if statement is used to execute a block of code if a specified condition is true.

## Syntax:

```
if (condition) {
  // Code to be executed if the condition is true
}
```

- condition: This is a Boolean expression that determines whether the code block inside the if statement should be executed.
- If the condition evaluates to true, the code block is executed; otherwise, it's skipped.

**Example:**

```
let age =27
if (age >= 18) {
  console.log("You are eligible for vote in Bharat.");
}
```

**Output:**

```
You are eligible for vote in Bharat.
```

## 2. if-else Statement:

- The if-else statement extends the if statement by allowing you to specify an alternative code block to be executed if the condition is false.

## Syntax:

```
if (condition) {
  // Code to be executed if the condition is true
} else {
  // Code to be executed if the condition is false
}
```

**Example:**

```
let num = 15;

if (num > 0) {
  console.log("The number is positive.");
} else {
  console.log("The number is not positive.");
}
```

**Output:**

```
number is positive.
```

**Example:**

```
//DYNAMIC INPUT IN node js
//Install the module prompt-sync
// npm i prompt-sync
const prompt = require("prompt-sync")();
```

```
let num = parseInt(prompt("Enter the Number:"))
if (num > 0) {
  console.log("The number is positive.");
} else {
  console.log("The number is not positive.");
}
```

**Output:**

```
Enter the Number:0
The number is not positive.

Enter the Number:-3
The number is not positive.

Enter the Number:15
The number is positive.
```

## 3. else if Ladder:

- else if ladder allows you to check multiple conditions sequentially and execute different code blocks based on the first condition that evaluates to true.
- It is used when you have multiple conditions to test.

## Syntax:

```
if (condition1) {
  // Code to be executed if condition1 is true
} else if (condition2) {
  // Code to be executed if condition2 is true
} else {
  // Code to be executed if none of the conditions are true
}
```

**Example:**

```
//DYNAMIC INPUT IN node js
//Install the module prompt-sync
// npm i prompt-sync
const prompt = require("prompt-sync")();
let score = parseInt(prompt("Enter the Marks:"))
if (score >= 90) {
  console.log("You got an A Grade.");
}
 else if (score >= 80) {
  console.log("You got a B Grad.");
}
else if (score >= 70) {
  console.log("You got a C Grad.");
}
else if (score >= 60) {
   console.log("You got a D Grad.");
  }
  else if (score >= 50) {
```

```
    console.log("You got a E Grad.");
  }
else {
  console.log("Fail!!!");
}
```

**Output:**

```
Enter the Marks:91
You got an A Grade.

Enter the Marks:59
You got a E Grad.

Enter the Marks:45
Fail!!!
```

## Switch Statement:

- Use the switch statement to select one of many code blocks to be executed.

**Syntax**

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

❖ **This is how it works:**

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.
- If there is no match, the default code block is executed.

**Example:**

**Question:** Program to print the correct week name as per given week number

```
1-Monday
2-Tuesday
3-Wednesday
4-Thursday
5-Friday
6-Saturday
7-Sunday
```

**Program:**

```
//DYNAMIC INPUT IN node js
//Install the module prompt-sync
```

```
// npm i prompt-sync
const prompt = require("prompt-sync")();
let week = parseInt(prompt("Enter Your Choice:"))
switch(week)
{
   case 1:
      console.log("Monday")
      break
   case 2:
      console.log("Tuesday")
      break
   case 3:
      console.log("Wednesday")
      break
   case 4:
      console.log("Thursday")
      break
   case 5:
      console.log("Friday")
      break
   case 6:
      console.log("Saturday")
      break
   case 7:
      console.log("Sunday")
      break
   default:
      console.log("Wrong input")  }
```

**output:**

```
Enter Your Choice:3
Wednesday
Enter Your Choice:6
Saturday
Enter Your Choice:10
Wrong input
```

# LOOP STATEMENT

- There are two major categories of loop -
  1. Entry controlled (for, while):- The loop body is executed if condition is true.
  2. Exit controlled(do-while):- The loop body executes first & then condition is checked.

**NOTE-** if condition is false, then loop body is not executed in case of for/while but will execute in do-while once

## ❖ For Loop:

- it is execute a block of code repeatedly for a specified number of times or until a certain condition is met.

### syntax:

```
for (initialization; condition; increment/decrement) {
  // Code to be executed in each iteration
}
```

- **Syntax Explanation:**
  - initialization: This part is typically used to initialize a variable before the loop starts. It's executed only once, at the beginning.
  - condition: This is the condition that is evaluated before each iteration. If the condition is true, the loop continues; if it's false, the loop stops.
  - increment/decrement: After each iteration of the loop, this part is used to update the loop control variable. It can be an increment (e.g., i++ to increase by 1) or a decrement (e.g., i-- to decrease by 1).
  - Code to be executed in each iteration: This is the block of code that is executed repeatedly as long as the condition remains true.

**Example:**

```
for (let i = 1; i <= 6; i++) {
  console.log(i);
}
```

**Output:**

```
1
2
3
4
5
6
```

**Explanation:**

let i = 1 initializes the loop control variable i to 1.
i <= 6 is the condition, and as long as i is less than or equal to 5, the loop continues.
i++ increments i by 1 after each iteration.
console.log(i) prints the value of i in each iteration.

## ❖ while Loop:

- The while loop repeatedly executes a block of code as long as a specified condition is true.

### syntax:

```
while (condition) {
  // Code to be executed as long as the condition is true
```

```
}
```

**Example:**

```
let count = 0;
while (count < 5) {
  console.log(count);
  count++;
}
```

**Output:**

```
0
1
2
3
4
```

## ❖ do...while Loop:

- The do...while loop is similar to the while loop, but it ensures that the block of code is executed at least once, even if the condition is false initially.

## Syntax:

```
do {
  // Code to be executed at least once
} while (condition);
Example:
let i=5;
do{
   console.log(i)
   i--
}while(i>=1)
do{
   console.log(i)
   i--
}while(i>6)
```

**output:**

```
5
4
3
2
1
0
```

## ❖ for in loop:

- for...in :- used for non-primitive data type
- The for...in loop is used to iterate over the properties of an object. It's often used for object iteration.

**Syntax:**

```
for (const property in object) {
  // Code to be executed for each property
}
```

**Example:**

```
let obj={name:"Sangam Kumar",age:15}
for(let k in obj){
   console.log(k,obj[k])
}
```

**output:**

```
Name Sangam Kumar
age 15
```

**Example:**

```
let arr=[10,20,"Red",50.2]
for(let i in arr)
   console.log(i,arr[i])
for(let i in arr){
   if(i==2)
      console.log(arr[i])
}
```

**output:**

```
0 10
1 20
2 Red
3 50.2
Red
```

**Example:**

```
let obj={Name:"Sangam Kumar",age:15}
for(let i in obj){
   if(obj[i]===15)
      console.log(obj[i])
}
```

**output:**

```
15
```

**Example:**

```
const person = {
   name: "Sangam Kumar",
   age: 15,
   job: "Developer"
 };
 for (const key in person) {
   console.log(key + ": " + person[key]);
 }
```

**Output:**

```
name: Sangam Kumar
age: 15
job: Developer
```

## ❖ for...of Loop:

- The for...of loop is used to iterate over elements of alterable objects like arrays, strings, and more.

**NOTE-** IF key & value both are required, use for..in otherwise IF value is required, use for..of

**Syntax:**

```
for (const element of iterable) {
  // Code to be executed for each element
}
```

**Example:**

```
let arr=[10,20,"Red",50.2]
let s="SKILLS"
for(let i of s)
   console.log(i)
```

**Output:**

```
S
K
I
L
L
S
```

**Example:**

```
const skills = ["Python", "HTML", "CSS","JavaScript","React js","Node JS"];
for (const i of skills) {
  console.log(i);
}
```

**output:**

```
Python
HTML
CSS
JavaScript
React js
Node JS
```

## ❖ forEach :

- The forEach() method calls a function for each element in an array.
- orEach() method is a convenient way to iterate over the elements of an array or any iterable object and perform an operation or execute a function for each element.

**Syntax:**

```
array.forEach(callback(currentValue[, index[, array]]) {
  // Code to be executed for each element
})
```

- array: The array (or any iterable object) that you want to iterate over.
- callback: A function that gets executed for each element in the array.
- currentValue: The current element being processed in the array.
- index (optional): The index of the current element being processed.
- array (optional): The array on which forEach() was called.

**Example:**

```
const colors = ['red', 'green', 'blue']
colors.forEach(function(color, index) {
  console.log(`Color at index ${index}: ${color}`)
})
```

**Output:**

Color at index 0: red
Color at index 1: green
Color at index 2: blue

Method-1

**Example:**

```
let arr=[1,2,3,4,"red"]
arr.forEach((i)=>console.log(i))
```

**output:**

1
2
3
4
red

**Method-2**

```
let arr=[1,2,3,4,"red"]
arr.forEach(
   function(i,j){
      console.log(`The value stored at index ${j} is ${i}`)
   }
)
```

**output:**

The value stored at index 0 is 1
The value stored at index 1 is 2
The value stored at index 2 is 3
The value stored at index 3 is 4
The value stored at index 4 is red

**NOTE-**

for..in & for..of are loops in JS
but forEach is a method

# EXERCISE:

**Problem-1**

- Program to check whether num is zero, positive or negative

```
//DYNAMIC INPUT IN node js
//Install the module prompt-sync
// npm i prompt-sync
const prompt = require("prompt-sync")();
 let num = parseInt(prompt("Num 1- "))
let num=4
//METHOD-1 USING IF-ELSE
if(num>0)
console.log("positive")
else if(num<0)
console.log("negative")
else
console.log("zero")
output:
positive
//METHOD-2 USING TERNARY OPERATOR
console.log(
   num>0 ? "positive"
   : num<0 ? "negative"
   : "zero"
)
```

**output:**

```
positive
```

## Problem-2

- Program to find the minimum of three numbers using dynamic input

```
const prompt = require("prompt-sync")();
let a = parseInt(prompt("Num 1- "))
let b = parseInt(prompt("Num 2- "))
let c = parseInt(prompt("Num 3- "))
if (a<b && a<c)
   console.log(`${a} is minimum`)
else if(b<c)
   console.log(`${b} is minimum`)
else
   console.log(`${c} is minimum`)
```

## Problem-3

- Program to take number as input & check whether it's +ve,-ve or zero

```
const prompt = require("prompt-sync")();
let a=parseInt(prompt("Enter the number: "))
if (a==0)
console.log("Zero")
else if (a>0)
```

```
console.log("+ve")
else
console.log("-ve")
```

## Problem-4

- Program to take 3 numbers as input & print them in descending order

```
const prompt = require("prompt-sync")();
let a = parseInt(prompt("Num 1- "))
let b = parseInt(prompt("Num 2- "))
let c = parseInt(prompt("Num 3- "))
if (a>b && a>c){
   if (b>c)
      console.log(a,b,c)
   else
      console.log(a,c,b)
}
else if(b>c){
   if (a>c)
      console.log(b,a,c)
   else
      console.log(b,c,a)
}
else{
   if (a>b)
      console.log(c,a,b)
   else
      console.log(c,b,a)
}
```

## Problem-5

- Program to print the factors of a given number

```
i/p-10
o/p- 1,2,5,10
const prompt = require("prompt-sync")();
let num=parseInt(prompt("Enter a number- "))
//using for loop
for(let i=1;i<=num;i++)
{
   if(num%i===0)
      console.log(i)
}
//using while loop
let i=1
while(i<=num){
   if(num%i===0)
      console.log(i)
   i++
}
```

## Problem-6

- Program to print the sum of factors of a given number

```
let sum=0
for(let i=1;i<=num;i++)
{
   if(num%i===0)
      sum+=i
}
console.log("Sum of factors is-",sum)
```

## Problem-7

- Program to check whether given num is prime or not

```
//Method-1
let ct=0
for(let i=1;i<=num;i++){
   if(num%i===0)
      ct = ct+1
}
if(ct==2)
   console.log(`${num} is prime`)
else
console.log(`${num} is not prime`)
//Method-2
let flag=true
if(num<=1)
   console.log(`${num} is not prime`)
else{
   for(let i=2;i<num;i++){
      if(num%i===0){
         flag=false
         break
      }
   }
   if(flag)
      console.log(`${num} is prime`)
   else
      console.log(`${num} is not prime`)
}
//factorial
const prompt = require("prompt-sync")();
let num=parseInt(prompt("Enter a number- "))
let f=1;
for(let i=1;i<=num;i++)
   f=f*i
console.log(`The factorial of ${num} is ${f}`)
```

## Problem-8

- Find the Gcd of two numbers

```
const prompt = require("prompt-sync")();
let num1=parseInt(prompt("Enter number1- "))
```

```
let num2=parseInt(prompt("Enter number2- "))
while(num2!=0){
   [num1,num2]=[num2,num1%num2]
}
console.log(num1)
```

## Problem-9

- Write the Multiplication table

```
const prompt = require("prompt-sync")();
let num=parseInt(prompt("Enter a number- "))
for(let i=1;i<=10;i++)
   console.log(`${num} x ${i} = ${num*i}`)
```

## Problem-10

- Find the Fibonacci series-- print the terms till n

```
const prompt = require("prompt-sync")();
let num=parseInt(prompt("Enter a number- "))
let f=-1
let s=1
let c
for(let i=1;i<=num;i++)
{
   c=f+s
   console.log(c)
   f=s
   s=c
}
```

**//for nth term in fibonacci series**

```
for(let i=1;i<=num;i++)
{
   c=f+s
   f=s
   s=c

}
console.log(c)
```

## Problem-11

- Program to print the count of each element from given array

```
let arr=[1,2,3,4,3,2,1,2,5,6,4,6,7,8,1]
let d={}
for(let i of arr){
   if(d[i]===undefined)
      d[i]=1
   else
      d[i]+=1
}
console.log(d)
```

## Problem-12

- Program to print the count of given element from given array

```
const prompt = require("prompt-sync")();
let arr=[1,2,3,4,3,2,1,2,5,6,4,6,7,8,1]
let ele=parseInt(prompt("Enter the element to be searched- "))
let ct=0
for(let i of arr){
   if(i===ele)
      ct+=1
}
console.log(`The count of ${ele} in given array is ${ct}`)
let arr=[1,2,3,4]
arr.reverse()
console.log(typeof arr,arr)
let ch=arr.toString()
console.log(typeof ch,ch,ch.length)
```

## Problem-13

- Program to reverse a string as per given o/p

i/p- "hello to all"
o/p- "all to hello"

```
let s="hello to all"
let r=s.split(" ")
r.reverse()
console.log(r.join(" "))
```

## Problem-14

- Program to reverse a string as per given o/p

i/p- "hello to all"
o/p- "olleh ot olleh"

```
let s="hello to all"
let r=s.split(" ")
let n=""
for(let i of r){
   i=i.split("")
   i.reverse()
   n=n+i.join("")+" "
}
console.log(n)
```

## Problem-15

- Program to reverse a string as per given o/p

i/p- "the sky is blue"
o/p- "the yks is eulb"

```
let s="the sky is blue"
let r=s.split(" ")
for(let i=0;i<r.length;i++){
   if(i%2!=0){

      r[i]=r[i].split("")
      r[i].reverse()
      r[i]=r[i].join("")
```

```
    }
  }
  console.log(r.join(" "))
```

## Problem-16

- Program to find nth prime number using function

```
function isPrime(n){
  let c=0
  for(let i=1;i<=n;i++){
    if(n%i===0)
      c=c+1
  }
  if(c===2)
    return true
  else
    return false
}
const prompt = require("prompt-sync")();
let num=parseInt(prompt("Enter number- "))
let c=0
let pn=2
while(c<=num){
  if(isPrime(pn)){
    c=c+1
    if(c===num)
      console.log(pn)
  }
  pn+=1
}
```

## Problem-17

- Recursive function to find gcd of two numbers

```
function gcd(a,b){
  if(b==0)
    return a
  else
    return gcd(b,a%b)
}
const prompt = require("prompt-sync")();
let num1=parseInt(prompt("Enter number1- "))
let num2=parseInt(prompt("Enter number2- "))
console.log(gcd(num1,num2))
```

## Problem-18

- Recursive function to find factorial of a number

```
function fact(num){
  if(num===1)
    return 1
  else
    return num*fact(num-1)
```

```
}
const prompt = require("prompt-sync")();
let n=parseInt(prompt("Enter number- "))
console.log(fact(n))
```

## Problem-19

- Recursive function for finding nth term of fibonacci series

```
function fib(n){
   if(n===1)
      return 0
   else if(n===2)
      return 1
   else
      return fib(n-1)+fib(n-2)
}
const prompt = require("prompt-sync")();
let n=parseInt(prompt("Enter number- "))
```

## Problem-20

- printing all terms of fibonacci series

```
for(let i=1;i<=n;i++)
   console.log(fib(i))
```

## Problem-21

take a number
if number is divisble by 3- print Fizz
if number is divisble by 5- print Buzz
if number is divisible by 3 and 5 both- print FizzBuzz
else- print "neither divisble by 3 nor 5"

```
let a=15
if (a%3===0 && a%5===0)
console.log("FizzBuzz")
else if (a%3===0)
console.log("Fizz")
else if (a%5===0)
console.log("Buzz")
else
console.log("neither divisible by 3 nor 5")
```

## Problem-22

- Swap the two number

1.**Using temp variable**

```
let a=20
let b=30
console.log(`Value of a is ${a} and value of b is ${b}`)
 let temp=a
 a=b
 b=temp
 console.log(`Value of a is ${a} and value of b is ${b}`)
```

2. **Using arithmetic operators(+,-),without temp var**

```
let a=20
```

```
let b=30
console.log(`Value of a is ${a} and value of b is ${b}`)
a=a+b
b=a-b
a=a-b
console.log(`Value of a is ${a} and value of b is ${b}`)
```

**method-3 ES6 feature**

```
let a=20
let b=30
console.log(`The value of a after swapping: ${a}`);
console.log(`The value of b after swapping: ${b}`);
[a, b] = [b, a];
console.log(`The value of a after swapping: ${a}`);
console.log(`The value of b after swapping: ${b}`);
```

**Method-4**

```
let a=20
let b=30
console.log(`Value of a is ${a} and value of b is ${b}`)
a=a*b
b=a/b
a=a/b
console.log(`Value of a is ${a} and value of b is ${b}`)
```

## Problem-23

- Program to find the minimum of three numbers using dynamic input

```
let a = parseInt(prompt("Num 1- "))
let b = parseInt(prompt("Num 2- "))
let c = parseInt(prompt("Num 3- "))
if (a<b && a<c)
   console.log(`${a} is minimum`)
else if(b<a && b<c)
   console.log(`${b} is minimum`)
else
   console.log(`${c} is minimum`)
```

# FUNCTIONS

- Function is a block of reusable code that performs a specific task or set of tasks.
- Functions are one of the fundamental building blocks of JavaScript and are used for organizing and structuring your code, making it more modular and easier to maintain.
- A JavaScript function is executed when "something" invokes it (calls it).

## ❖ Advantage of JavaScript function:

- There are mainly two advantages of JavaScript functions.
- Code reusability: We can call a function several times so it saves coding.
- Less coding: We don't need to write many lines of code each time to perform a common task.

## Syntax:

- **Function Declaration:**

```
function functionName(parameters) {
  // Function body
  // Code to be executed when the function is called
}
```

- **functionName:** This is the name of the function.
- **parameters:** These are the input values (arguments) that the function can accept. You can have zero or more parameters, separated by commas.
- **Function body:** This is where you define code that gets executed when the function is called.

## ❖ User defined Function:

**Example:**

```
function myFun()//Function definition
{
  //content of function is called function body
  console.log("Welcome to Twksaa skills center")
}
myFun()//calling a function
```

**Output:**

```
Welcome to Twksaa skills center
```

## ❖ Use of return in function-

It is used to store the value returned in function

## Example-1:

```
function add(a, b)//a, b are parameters
{
  return a+b
}
function add2(a,b)
{
  console.log("Function output",a+b)
}
console.log(add("Sum=",10,20))//10,20 are arguments
add2(15,20)
```

**Output:**

```
Sum=10
Function output 35
```

**Example-2:**

```
function add(a,b)//a,b are parameters
{
   return a+b
}
function add2(a,b)
{
   console.log("Function output",a+b)
}
console.log(add(10,20))//10,20 are arguments
add2(30,40)
let r1=add(10,20)
let r2=add2(3,4)
console.log(r1)
console.log("Value of r2 is",r2)
```

**Output:**

```
Function output 70
Function output 7
30
Value of r2 is undefined
```

## ❖ Function Expression:

- A JavaScript function can also be defined using an expression.
- A function expression can be stored in a variable:

**Syntax:**

```
const functionName = function(parameters) {
   // Function body
   // Code to be executed when the function is called
}
```

- **functionName:** This is an optional name for the function. You can omit it if you want an anonymous function.
- **parameters:** These are the input values (arguments) that the function can accept.
- **Function body:** This is where you define the code that gets executed when the function is called.

**Example-1:**

```
const add = function(a, b) {
   return a + b;
}
r1=add(15,20)
console.log(r1)
r2=add(25,50)
console.log(r2)
r3=add(50,53)
console.log(r3)
```

**output:**

```
35
75
103
```

**Example-2:**

```
const x = function (a, b)
 {
return a * b
}
let z = x(3, 6)
console.log(z)
```

**Output:**

```
7
```

**Note:** above function is actually an anonymous function (a function without a name). Functions stored in variables do not need function names. They are always invoked (called) using the variable name.

## ❖ Function() Constructor:

- Function constructor is a built-in function that can be used to create new function objects dynamically.
- It allows you to create functions at runtime by providing a list of arguments and a function body as strings.

**Syntax:**

- new Function(arg1, arg2, ..., functionBody)

Example:

```
// Create a simple function that adds two numbers
const addFunction = new Function('a', 'b', 'return a + b;');
// Call the dynamically created function
const result = addFunction(3, 6);
console.log(result);
```

**Output:**

9

**Explanation:**

- addFunction is created using the Function constructor.
- It takes two arguments, 'a' and 'b', and returns their sum.

## ❖ Function Parameters and Arguments:

- Function parameters are the names listed in the function definition.
- Function arguments are the real values passed to (and received by) the function.
- Parameters are the placeholders in the function definition, while arguments are the actual values passed when calling the function.

**Syntax:**

```
function functionName(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

## ➢ Parameter Rules:

- JavaScript function definitions do not specify data types for parameters.
- JavaScript functions do not perform type checking on the passed arguments.
- JavaScript functions do not check the number of arguments received.

**Example:**

```
function greet(name) //here name is parameter
{
   console.log(`Hello, ${name}!`);
 }
 greet("Sangam Kumar");  //here Sangam Kumar is argument
 greet(); // "Hello, undefined!"
```

**Output:**

```
Hello, Sangam Kumar!
Hello, undefined!
```

**Note:**

- If a function is called with missing arguments (less than declared), the missing values are set to undefined.

## ❖ Default Parameters:

- ES6 allows function parameters to have default values.
- You can specify default values for function parameters, which are used if argument is not provided

**Example-1:**

```
function greet(name = "Skills") //Here name="skills" is a Default Parameters:
{
   console.log(`TWKSAA, ${name}!`);
 }
 greet();
 greet("Center");
```

**Output:** TWKSAA, Skills!
TWKSAA, Center!

**Example-2:**

- If y is not passed or undefined, then y = 15.

```
function myFunction(x, y = 15) {
 return x + y
}
let r= myFunction(20)
console.log(r)
```

**Output:** 35

## ❖ Rest Parameters:

- The rest parameter allows a function to accept an arbitrary number of arguments as an array.
- It is denoted by three dots (...) followed by a parameter name.
- The rest parameter (...) allows a function to treat an indefinite number of arguments as an array:

**Example-1:**

```
function sum(...numbers) {
 return numbers.reduce((total, num) => total + num, 0)
}
const result = sum(1, 2, 3, 4)
console.log(result)
```

**Output:** 10

**Example-2:**

```
function sum(...args) {
 let sum = 0
 for (let arg of args) sum += arg
 return sum
}
let A = sum(3,6, 9, 16, 25, 29, 100, 66, 77,8)
console.log(A)
```

**Output:** 339

## ❖ Arguments are Passed by Value:

- The parameters, in a function call, are the function's arguments.
- JavaScript arguments are passed by value: The function only gets to know the values, not the argument's locations.
- If a function changes an argument's value, it does not change parameter's original value.
- Changes to arguments are not visible (reflected) outside the function.

# ARROW FUNCTION

- Arrow functions provide a concise way to write functions, especially for simple one-liners. They automatically return the expression result.

**syntax:**

arrow function is defined using the => (fat arrow)

```
const functionName = (parameters) => {
   // Function body
   // Code to be executed when the function is called
}
```

- **FunctionName:** This is an optional name for the function, just like in function expressions.
- **parameters:** These are the input values (arguments) that the function can accept.
- **Function body:** This is where you define the code that gets executed when the function is called.
  - **Arrow function {()=>}** is concise way of writing Javascript functions in shorter way. Arrow functions were introduced in the ES6 version.

**Method-1**

```
const add3=(a,b)=>console.log(a+b)
add3(10,20)
```

**output:** 30

**Method-2**

```
let add3=(x,y)=>x+y
Here add3 is function name &  x, y are parameters
console.log(add3(10,20))
Recursion-When a function calls itself, it is recursion
```

**Example:**

```
// Arrow function without parameters
const greet = () => {
  console.log("Hello!");
}
// Arrow function with a single parameter
const square = (x) => {
  return x * x;
}
// Arrow function with multiple parameters
const add = (a, b) => {
  return a + b;
}
// Arrow function with a concise body (implicit return)
const double = (x) => x * 2;
```

**Example:**

```
function Person(name) {
   this.name = name;
   // Regular function with a different "this" context
   this.sayHello = function () {
     console.log(`Hello, ${this.name}!`);
   };
```

```
  // Arrow function with the same "this" context
  this.sayHi = () => {
    console.log(`Hi, ${this.name}!`);
  };
 }
const person = new Person("Sangam Kumar");
person.sayHello(); // Outputs "Hello, Sangam Kumar!"
person.sayHi();    // Outputs "Hi, Sangam Kumar!"
```

**Output:**

```
Hello, Sangam Kumar!
Hi, Sangam Kumar!
```

## Implicit Return:

- Arrow functions allow for implicit return when the function body consists of a single expression.
- In such cases, you can omit the curly braces {} and the return keyword. The result of the expression is automatically returned.

**Example:**

```
const square = (x) => x * x; // Implicit return
const add = (a, b) => {
  return a + b; // Explicit return
};
```

## ❖ When to Use Arrow Functions:

- Arrow functions are most suitable for short, concise functions, especially those used as callbacks or for simple calculations.
- They are not a replacement for regular functions in all scenarios, as regular functions still have their place when you need the flexibility of this keyword, function hoisting, or named functions.

# FUNCTION INVOCATION

- function invocation is the process of calling a function to execute its code.
- To execute a function, you need to invoke (call) it by using its name followed by parentheses.
- If the function has parameters, you pass arguments inside the parentheses.

## ❖ Why function invocation needs?

- The code inside a function is not executed when the function is defined.
- The code inside a function is executed when the function is invoked.
- It is common to use the term "call a function" instead of "invoke a function".

## ❖ How many ways you can call the function or function is invoked?

- The following ways you can call a function.
    1) Function Declaration
    2) Function Expression
    3) Arrow Function
    4) Method Invocation
    5) Constructor Invocation
    6) Function.call()
    7) Function.apply()
    8) Function Invocation without Arguments
    9) Immediately Invoked Function Expression (IIFE)
    10) Callback Functions

## 1. Function Declaration:

- You can define a function using the `function` keyword and then call it by its name.

**Example:**

```
function greet(name) {
   console.log(`Hello, ${name}!`);
  }
  greet("Sangam Kumar"); // Calling a function declared using function declaration
```

## 2. Function Expression:

- You can define a function as an expression and then call it. This is often used for anonymous functions or functions assigned to variables.

**Example:**

```
const sayHello = function (name) {
console.log(`Hello, ${name}!`);
 };
sayHello("Satyam"); // Calling a function defined as an expression
```

## 3. Arrow Function:

- Arrow functions provide a concise way to define and call functions.

**Example:**

```
const multiply = (a, b) => a * b;
const result = multiply(4, 5); // Calling an arrow function
```

## 4. Method Invocation:

- When a function is a property of an object, you can invoke it using dot notation.

**Example:**

```
const person = {
```

```
  name: "Bob",
  greet: function () {
    console.log(`Hello, ${this.name}!`);
  },
};
person.greet(); // Calling a method of an object
```

## 5. Constructor Invocation:

- You can create instances of objects using constructor functions and the `new` keyword.

Example:

```
function Dog(name) {
        this.name = name;
                }
const dog1 = new Dog("Rover"); // Calling a constructor function
```

## 6. Function.call() and Function.apply():

❖ These methods allow you to explicitly invoke a function and specify the value of `this` and pass arguments.

**Example:**

```
function introduce() {
        console.log(`Hello, I am ${this.name}.`);
        }
const person = { name: "Sangam" };
introduce.call(person); // Using call
introduce.apply(person); // Using apply
```

## 7. Function Invocation without Arguments:

❖ Functions can also be invoked without passing any arguments by using empty parentheses `()`.

## 8. Immediately Invoked Function Expression (IIFE):

❖ An IIFE is a function that is defined and invoked immediately. It's often used to create a private scope for variables.

```
(function () {
        // Code here
})();
```

## 9. Callback Functions:

❖ You can pass functions as arguments to other functions and call them later.

**Example:**

```
function performOperation(x, y, operation) {
        return operation(x, y);
        }
function add(a, b) {
return a + b;
}
const result = performOperation(3, 4, add); // Calling a function as a callback
```

- const result = add(3, 4); // Calling the add function
- Return Statement:
- Functions can return values using the return statement. The returned value can be used wherever the function is called. If a function doesn't specify a return value, it implicitly returns undefined.

**Example:**

```
function subtract(a, b) {
  return a - b;
}
const difference = subtract(8, 3); // difference will be 5
```

## Function Scope:

- Variables declared inside a function are locally scoped, meaning they are only accessible within that function. This concept is known as "function scope."

## Example:

```
function scopeExample() {
  const localVar = "I am local";
  console.log(localVar); // Accessible within the function
}
scopeExample();
console.log(localVar); // Error: localVar is not defined
```

**Function Hoisting:**

- Function declarations are hoisted, meaning they can be called before they are defined in the code. This is not the case with function expressions.

**Example:**

```
hoistedFunction(); // This works
function hoistedFunction() {
  console.log("I was hoisted.");
}
nonHoistedFunction(); // Error: nonHoistedFunction is not a function
const nonHoistedFunction = function () {
  console.log("I was not hoisted.");
};
```

## ❖ Closures:

- A closure is a function that remembers the variables from the outer (enclosing) function's scope even after the outer function has finished executing. Closures are often used to create private variables and maintain state.

**Example:**

```
function createCounter() {
  let count = 0;
  return function () {
    count++;
    return count;
  };
}
const counter = createCounter();
console.log(counter()); // 1
console.log(counter()); // 2
```

## ❖ Function Declarations vs. Expressions:

- Function declarations are hoisted and can be called before they are defined. Function expressions are not hoisted and can only be called after they are defined.

```
// Function Declaration
```

```
hoistedFunction(); // Works
function hoistedFunction() {
  console.log("I was hoisted.");
}
// Function Expression
nonHoistedFunction(); // Error: nonHoistedFunction is not a function
const nonHoistedFunction = function () {
  console.log("I was not hoisted.");
};
```

# Higher-Order Functions

- A higher-order function is a function that takes one or more functions as arguments or returns a function as its result. Higher-order functions are a powerful concept in JavaScript, enabling functional programming techniques.

**Example:**

```
function operateOnArray(array, operation) {
  const result = [];
  for (const item of array) {
    result.push(operation(item));
  }
  return result;
}
const numbers = [1, 2, 3, 4];
const doubledNumbers = operateOnArray(numbers, (num) => num * 2);
console.log(doubledNumbers); // [2, 4, 6, 8]
```

**Example:**

```
function add(a,b){
  console.log("add",a+b)
};
function sub(a,b){
  console.log("sub",a-b)
}

function hof(a,b,fn){
  fn(a,b) \\ function in side function with arguments
};
hof(2,4,add)
```

# Anonymous Functions

- Functions that don't have a name are called anonymous functions. They are often used as arguments to other functions or as immediately invoked function expressions (IIFE).

**Example:**

```
const greeting = function (name) {
  console.log(`Hello, ${name}!`);
};
greeting("Alice"); // "Hello, Alice!"
// IIFE (Immediately Invoked Function Expression)
(function () {
  console.log("I am an IIFE.");
})();
```

## ❖ Callbacks and Asynchronous JavaScript:

- JavaScript commonly uses callbacks to handle asynchronous operations like reading files or making network requests. A callback is a function passed as an argument to another function and is executed later when a specific event occurs.

**Example:**

```
function fetchData(callback) {
  setTimeout(() => {
    const data = "Some async data";
    callback(data);
  }, 1000);
}
function process(data) {
  console.log(`Processing: ${data}`);
}
fetchData(process); // Executes process when data is ready
```

## ❖ Function Scopes:

- JavaScript has two main types of function scope: global scope (accessible from anywhere) and local scope (limited to the function where a variable is declared).

```
// Global scope
const globalVar = "I am global";
function scopeExample() {
  // Local scope
  const localVar = "I am local";
  console.log(globalVar); // Accessible in local scope
}
scopeExample();
console.log(globalVar
```

# OBJECT:

- In JavaScript, almost "everything" is an object.
  - Booleans can be objects (if defined with the new keyword)
  - Numbers can be objects (if defined with the new keyword)
  - Strings can be objects (if defined with the new keyword)
  - Dates are always objects
  - Maths are always objects
  - Regular expressions are always objects
  - Arrays are always objects
  - Functions are always objects
  - Objects are always objects
  - All JavaScript values, except primitives, are objects.
- javaScript object is an entity having state and behavior (properties and method).

**Example:** car, Mobile, pen etc.

- JavaScript is an object-based language. Everything is an object in JavaScript.
- JavaScript is template based not class based. (Means don't create class to get the object. But, we direct create objects.)

**Note:** Objects written as name value pairs are similar to:

- Associative arrays in PHP
- Dictionaries in Python
- Hash tables in C
- Hash maps in Java
- Hashes in Ruby and Perl

## ❖ Object Methods:

- Methods are actions that can be performed on objects.
- Object properties can be both primitive values, other objects, and functions.
- An object method is an object property containing a function definition.

## ❖ Creating Objects:

- There are following ways to create objects.

1. Object Literal: Using {} to define an object with properties and values.
2. Constructor Functions: Creating objects using functions with the new keyword.
3. ES6 Class Syntax: Using the class syntax introduced in ECMAScript 2015 (ES6).
4. Object.create(): Creating objects with a specified prototype using Object.create().
5. Factory Functions: Creating objects with regular functions that return objects.
6. Singleton Objects: Creating a single instance of an object throughout the application.
7. Using the new Operator with Built-in Constructors: Creating objects of built-in types like Array, Date, and RegExp using new.

## Example:

### 1.Object Literal:

```
let person = {
  firstName: "Sangam",
  lastName: "Kumar",
};
```

## 2.Constructor Functions:

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}
let person = new Person("Satyam", "Kumar");
```

## 3.ES6 Class Syntax:

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}
let person = new Person("Sangam", "Kumar");
```

## 4.Object.create():

```
let personPrototype = {
  firstName: "",
  lastName: "",
};
let person = Object.create(personPrototype);
person.firstName = "Sangam";
person.lastName = "Kumar";
```

## 5. Factory Functions:

```
function createPerson(firstName, lastName) {
  return {
    firstName: firstName,
    lastName: lastName,
  };
}
let person = createPerson("Sangam", "Kumar");
```

## 6.Singleton Objects:

```
let singleton = (() => {
  let instance;
  function createInstance() {
    return {
      // Properties and methods go here
    };
  }
  return {
    getInstance: function() {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    },
  };
```

```
})();
let obj1 = singleton.getInstance();
let obj2 = singleton.getInstance();
console.log(obj1 === obj2); // true
```

## 7.Using the new Operator with Built-in Constructors:

```
let myArray = new Array(1, 2, 3);
let today = new Date();
let regex = new RegExp("pattern");
```

### ❖ 1.Object Literal:

- The simplest and most commonly used way to create an object is by using the object literal syntax. You define an object and its properties within curly braces {}.

**Example:**

```
// Creating an object using Object Literal
let person = {
  firstName: "Sangam",
  lastName: "Kumar",
  age: 15, }
// Accessing properties of the object
console.log(person.firstName) // Output: Sangam
console.log(person.lastName) // Output: Kumar
console.log(person.age)      // Output: 15
```

**Output:**

```
Sangam
Kumar
15
```

**Note:** in this example, we've created an object named person with three properties: firstName, lastName, and age.

### ❖ 2.Constructor Functions:

- You can create objects in JavaScript using constructor functions.
- Constructor functions are regular functions that are intended to be used with the new keyword to create instances of objects.

**Example:**

```
// Constructor function to create Person objects
function Person(firstName, lastName, age) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.age = age;
}
// Creating instances of the Person object using the constructor
let person1 = new Person("Sangam", "Kumar", 15);
let person2 = new Person("Satyam", "Kumar", 20);
// Accessing properties of the objects
console.log(person1.firstName); // Output: Sangam
console.log(person1.lastName);  // Output: Kumar
console.log(person1.age);       // Output: 15
console.log(person2.firstName); // Output: Satyam
```

```
console.log(person2.lastName);  // Output: Kumar
console.log(person2.age);       // Output: 20
```

**Output:**

```
Sangam
Kumar
15
Satyam
Kumar
20
```

**Explanation:**

- We define a constructor function Person with parameters firstName, lastName, and age.
- Inside the constructor function, we use the this keyword to assign values to the object's properties based on the provided arguments.
- We create two instances of the Person object, person1 and person2, using the new keyword followed by the constructor function.

## ❖ 3.ES6 Class Syntax:

- You can create objects in JavaScript using ES6 class syntax.
- Classes provide a more structured and familiar way to define object blueprints with constructors and methods

**Example:**

```
// ES6 class to create Person objects
class Person {
  constructor(firstName, lastName, age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
  }
  sayHello() {
    console.log(`Hello, my name is ${this.firstName} ${this.lastName}.`);
  }
}
// Creating instances of the Person class
let person1 = new Person("Sangam", "Kumar", 15);
let person2 = new Person("Satyam", "Kumar", 20);
// Accessing properties and calling a method of the objects
console.log(person1.firstName); // Output: Sangam
console.log(person1.lastName);  // Output: Kumar
console.log(person1.age);       // Output: 15
console.log(person2.firstName); // Output: Satyam
console.log(person2.lastName);  // Output: Kumar
console.log(person2.age);       // Output: 20
person1.sayHello(); // Output: Hello, my name is Sangam Kumar.
person2.sayHello(); // Output: Hello, my name is Satyam Kumar.
```

**Output:**

```
Sangam
Kumar
15
```

```
Satyam
Kumar
20
Hello, my name is Sangam Kumar.
Hello, my name is Satyam Kumar.
```

Explanation.

- We define a class Person using the class keyword, which includes a constructor method to initialize object properties and a sayHello method.
- Inside the constructor method, we use the this keyword to assign values to the object's properties based on the provided arguments.
- We create two instances of the Person class, person1 and person2, using the new keyword followed by the class name.

## ❖ 4.Object.create():

- You can create objects in JavaScript using the Object.create() method.
- This method allows you to create an object with a specified prototype.

**Example:**

```
// Create a prototype object
let personPrototype = {
 greet: function () {
   console.log(`Hello, my name is ${this.firstName} ${this.lastName}.`);
 },
};
// Create an object with the specified prototype
let person = Object.create(personPrototype);
// Add properties to the object
person.firstName = "Sangam";
person.lastName = "Kumar";
// Access properties and call the method of the object
console.log(person.firstName); // Output: Sangam
console.log(person.lastName);  // Output: Kumar
person.greet(); // Output: Hello, my name is Sangam Kumar.
```

**Explanation:**

- We create a prototype object called personPrototype that contains a greet method.
- We create a new object called person using Object.create(personPrototype). This means that person inherits the properties and methods of personPrototype.
- We add properties firstName and lastName to the person object.

## ❖ 5.Factory Functions:

- You can create objects in JavaScript using factory functions,
- which are regular functions that return new objects.

**Example:**

```
// Factory function to create a person object
function createPerson(firstName, lastName, age) {
 return {
   firstName: firstName,
   lastName: lastName,
   age: age,
   sayHello: function() {
```

```
      console.log(`Hello, my name is ${this.firstName} ${this.lastName}.`);
    }
  };
}
// Create instances of the person object using the factory function
let person1 = createPerson("Sangam", "Kumar", 15);
let person2 = createPerson("Satyam", "Kumar", 20);
// Access properties and call the method of the objects
console.log(person1.firstName); // Output: Sangam
console.log(person1.lastName);  // Output: Kumar
console.log(person1.age);       // Output: 15
console.log(person2.firstName); // Output: Satyam
console.log(person2.lastName);  // Output: Kumar
console.log(person2.age);       // Output: 20
person1.sayHello(); // Output: Hello, my name is Sangam Kumar.
person2.sayHello(); // Output: Hello, my name is Satyam Kumar.
```

**Output:**

```
PS D:\js rev\rajt3> node ./p1.js
Sangam
Kumar
15
Satyam
Kumar
20
Hello, my name is Sangam Kumar.
Hello, my name is Satyam Kumar.
```

Explanation:

- We define a factory function createPerson that takes firstName, lastName, and age as parameters and returns an object.
- Inside the factory function, we create an object with properties firstName, lastName, and age, along with a sayHello method.
- We create two instances of the person object using the createPerson factory function.

## ❖ 6.Singleton Objects:

- you can create singleton objects using various patterns.
- One common way is to use an immediately invoked function expression (IIFE) to encapsulate your object creation and ensure that only one instance of the object is created.

**Example:**

```
let singleton = (() => {
 // Private variables and functions can be defined here
 let instance;
 // Private initialization logic
 function init() {
  return {
   firstName: "Sangam",
   lastName: "Kumar",
   age: 15,
   sayHello: function() {
```

```
        console.log(`Hello, my name is ${this.firstName} ${this.lastName}.`);
      },
    };
  }
  return {
    // Public method to get the instance of the object
    getInstance: function() {
      if (!instance) {
        instance = init();
      }
      return instance;
    },
  };
})();
// Creating instances of the singleton object
let obj1 = singleton.getInstance();
let obj2 = singleton.getInstance();
// Accessing properties and calling the method of the objects
console.log(obj1.firstName); // Output: Sangam
console.log(obj1.lastName);  // Output: Kumar
console.log(obj1.age);       // Output: 15
console.log(obj2.firstName); // Output: Sangam
console.log(obj2.lastName);  // Output: Kumar
console.log(obj2.age);       // Output: 15
obj1.sayHello(); // Output: Hello, my name is Sangam Kumar.
obj2.sayHello(); // Output: Hello, my name is Sangam Kumar.
// Checking if obj1 and obj2 are the same instance
console.log(obj1 === obj2); // Output: true
```

**Output:**

```
Sangam
Kumar
15
Sangam
Kumar
15
Hello, my name is Sangam Kumar.
Hello, my name is Sangam Kumar.
true
```

Explanation:

- We create a singleton object using an IIFE (immediately invoked function expression) to encapsulate the object creation logic.
- Inside the IIFE, we define a private variable instance to hold the single instance of the object.
- The init function initializes the object with properties and methods.
- We expose a public method getInstance that checks if an instance already exists. If it doesn't, it creates a new instance using the init function and returns it. If an instance already exists, it returns the existing instance.

- We demonstrate that obj1 and obj2 are the same instance by comparing them with obj1 === obj2.

## ❖ 7.Using the new Operator with Built-in Constructors:

- you can create objects using the new operator with built-in constructors like Object, Array, Date, and RegExp.

**Example:**

Creating an Object:

```
// Using the Object constructor to create an empty object
let person = new Object();
// Adding properties to the object
person.firstName = "Sangam";
person.lastName = "Kumar";
// Accessing properties of the object
console.log(person.firstName); // Output: Sangam
console.log(person.lastName);  // Output: Kumar
```

Creating an Array:

```
// Using the Array constructor to create an array
let numbers = new Array(1, 2, 3, 4, 5);
// Accessing elements of the array
console.log(numbers[0]); // Output: 1
console.log(numbers[2]); // Output: 3
```

Creating a Date Object:

```
// Using the Date constructor to create a Date object
let today = new Date();
// Accessing date properties and methods
console.log(today.getFullYear()); // Output: 2023
console.log(today.getMonth());    // Output: 9
```

Creating a Regular Expression (RegExp):

```
// Using the RegExp constructor to create a regular expression
let regex = new RegExp("pattern");
// Testing the regular expression against a string
let result = regex.test("This is a pattern.");
console.log(result); // Output: true
```

## ❖ JavaScript Properties:

- Properties are the values associated with a JavaScript object.
- A JavaScript object is a collection of unordered properties.
- Properties can usually be changed, added, and deleted, but some are read only.

### ❖ Accessing JavaScript Properties:

The syntax for accessing the property of an object is:

```
objectName.property     // person.age
or
objectName["property"]   // person["age"]
or
objectName[expression]   // x = "age"; person[x]
```

The expression must evaluate to a property name.

# ACCESSING DATA FROM OBJECT

- there are following way you can access data objects:
    1) Dot Notation
    2) Bracket Notation
    3) Computed Property Names (ES6)
    4) Object Destructuring
    5) For...In Loop
    6) Object Methods (e.g., Object.keys(), Object.values(), Object.entries())

## 1.Dot Notation:

**Example:**

```
// Creating an object
let person = {
   firstName: "Sangam",
   lastName: "Kumar",
   age: 15,
 };
 // Accessing data using dot notation
 console.log(person.firstName); // Output: Sangam
 console.log(person.lastName);  // Output: Kumar
 console.log(person.age);      // Output: 15
```

**Output:**

```
Sangam
Kumar
15
```

**Explanation:**

- We create an object named person with properties firstName, lastName, and age.
- To access the data within the object, we use dot notation, where person.firstName accesses the firstName property, person.lastName accesses the lastName property, and person.age accesses the age property.

## 2.Bracket Notation:

- You can access data from an object in JavaScript using bracket notation by specifying the object's name followed by square brackets [ ] and placing the property name you want to access inside the brackets as a string.

**Example:**

```
// Creating an object
let person = {
   firstName: "Sangam",
   lastName: "Kumar",
   age: 15,
 };
 // Accessing data using bracket notation
 console.log(person["firstName"]); // Output: Sangam
 console.log(person["lastName"]);  // Output: Kumar
 console.log(person["age"]);      // Output: 15
```

**output:**

```
Sangam
Kumar
15
```

**Explanation:**

- We create an object named person with properties firstName, lastName, and age.
- To access the data within the object using bracket notation, we place the property name inside square brackets as a string, such as person["firstName"], person["lastName"], and person["age"].

## 3.Computed Property Names (ES6):

```
// Creating an object with computed property names
let propertyName = "firstName";
let person = {
  [propertyName]: "Sangam",
  ["last" + "Name"]: "Kumar",
  [3 + 3]: 6,
};
// Accessing data using computed property names
console.log(person[propertyName]);   // Output: Sangam
console.log(person["lastName"]);     // Output: Kumar
console.log(person[3 + 3]);          // Output: 6
```

**output:**

```
Sangam
Kumar
6
```

**Explanation:**

- We create an object named person with properties whose names are computed dynamically using computed property names.
- The propertyName variable is used to define the property name as "firstName".
- We also use expressions like ["last" + "Name"] and [2 + 2] to compute property names at runtime.
- When accessing the data, we use square brackets [ ] with the computed property names to retrieve the corresponding values.

## 4.Object Destructuring:

- You can access data from an object in JavaScript using object destructuring.
- Object destructuring allows you to extract specific properties from an object and assign them to variables with the same names as the properties.

**Example:**

```
// Creating an object
let person = {
   firstName: "Sangam",
   lastName: "Kumar",
   age: 15,
 };
 // Using object destructuring to access data
```

```
let { firstName, lastName, age } = person;
// Accessing data using the destructured variables
console.log(firstName); // Output: Sangam
console.log(lastName);  // Output: Kumar
console.log(age);       // Output: 15
```

**output:**

```
Sangam
Kumar
15
```

**Explanation:**

- We create an object named person with properties firstName, lastName, and age.
- We use object destructuring to extract the properties firstName, lastName, and age from the person object and assign their values to corresponding variables.
- The variables firstName, lastName, and age now hold the values extracted from the object, and we can access them directly.

## 5.For...In Loop:

- You can access data from an object in JavaScript using a for...in loop. This loop iterates through the properties of an object and allows you to access their values.

**Example:**

```
// Creating an object
let person = {
   firstName: "Satyam",
   lastName: "Kumar",
   age: 20,
 };

 // Using a for...in loop to access data
 for (let key in person) {
   console.log(`${key}: ${person[key]}`);
 }
```

**Output:**

```
firstName: Satyam
lastName: Kumar
 age: 20
```

**Explanation:**

- We create an object named person with properties firstName, lastName, and age.
- We use a for...in loop to iterate through the properties of the person object.
- Inside the loop, we access the current property's name using the key variable and its corresponding value using person[key].

## 6.Object Methods:

- You can access data from an object in JavaScript using various object methods such as Object.keys(), Object.values(), and Object.entries().
- These methods return arrays containing the keys, values, or key-value pairs (entries) of an object, respectively.
- Using Object.keys():

**Example:**

```
// Creating an object
let person = {
   firstName: "Sangam",
   lastName: "Kumar",
   age: 15,
  };
  // Accessing data using Object.keys()
  let keys = Object.keys(person);
  // Printing the keys
  console.log(keys); // Output: ["firstName", "lastName", "age"]
```

**Output:**

```
[ 'firstName', 'lastName', 'age' ]
Using Object.values():
```

## ❖ Adding New Properties:

- You can add new properties to an existing object by simply giving it a value.
- Assume that the person object already exists - you can then give it new properties:

**Example:**

- person.nationality = "Indian";

## ❖ Deleting Properties:

- The delete keyword deletes a property from an object:

**Example:**

```
const person = {
  firstName: "Sangam",
  lastName: "Kumar",
  age: 50,
  eyeColor: "blue"
};
delete person.age;
```

**Or:**

```
const person = {
  firstName: "Sangam",
  lastName: "Kumar",
  age: 50,
  eyeColor: "blue"
};
delete person["age"];
```

Note:

- The delete keyword deletes both the value of the property and the property itself.
- After deletion, the property cannot be used before it is added back again.
- The delete operator is designed to be used on object properties. It has no effect on variables or functions.
- The delete operator should not be used on predefined JavaScript object properties. It can crash your application.

## ❖ Nested Objects:

- Values in an object can be another object:

**Example:**

```
let person={
   name:"Sangam",
   age:25,
   fav:["red","blue","green"],
   marks:{
      maths:50,
      sci:70,
      hindi:90
   }
}
console.log(person)
console.log(person["name"])//Access value in object via key- method 1
console.log(person.name)//Access value in object via key- method 2
console.log(person.fav[2],person.fav[0],person.fav[1])
console.log(person.marks.sci)
console.log(person["marks"]["sci"]) //Nested Objects, Values in an object be in another object
```

**Output:**

```
{
 name: 'Sangam',
 age: 25,
 fav: [ 'red', 'blue', 'green' ],
 marks: { maths: 50, sci: 70, hindi: 90 }
}
Sangam
Sangam
green red blue
70
70
```

## ❖ Nested Arrays and Objects:

- Values in objects can be arrays, and values in arrays can be objects:

**Example:**

```
const myObj = {
 name: "John",
 age: 30,
 cars: [
  {name:"Ford", models:["Fiesta", "Focus", "Mustang"]},
  {name:"BMW", models:["320", "X3", "X5"]},
  {name:"Fiat", models:["500", "Panda"]}
 ]
}
```

# STRING:

- A JavaScript string is zero or more characters written inside quotes.

**Example:**

```
let text = "skills"
```

- You can use single or double quotes:

**Example:**

```
let Name1 = "wit"  // Double quotes
let Name2 = 'RID'  // Single quotes
```

- You can use quotes inside a string, as long as they don't match the quotes surrounding the string:

**Example:**

```
let answer1 = "It's alright";
let answer2 = "He is called 'Hi'"
let answer3 = 'He is called "Hello"'
```

## ❖ Escape Character:

| Code | Result | Description |
|---|---|---|
| \' | ' | Single quote |
| \" | " | Double quote |
| \\ | \ | Backslash |
| \b | | Backspace |
| \f | | Form Feed |
| \n | | New Line |
| \r | | Carriage Return |
| \t | | Horizontal Tabulator |
| \v | | Vertical Tabulator |

## ❖ String Methods:

1) String length
2) String slice()
3) String substring()
4) String substr()
5) String replace()
6) String replaceAll()
7) String toUpperCase()
8) String toLowerCase()
9) String concat()
10) String trim()
11) String trimStart()
12) String trimEnd()
13) String padStart()
14) String padEnd()
15) String charAt()
16) String charCodeAt()
17) String split()

## 1. String length (`length` property):

- Returns the length of the string.

**Example:**

```
const str = "Hello, World!";
console.log(str.length);
```

**Output:** 13

## 2. String slice(start, end):

- Extracts a portion of the string and returns it as a new string.

**Example:**

```
const str = "Hello, World!";
const slicedStr = str.slice(0, 5);
console.log(slicedStr);
```

**Output:** "Hello"

## 3. String substring(start, end):

- Similar to `slice()`, it extracts a portion of the string and returns it as a new string.

**Example:**

```
const str = "Hello, World!";
const subStr = str.substring(7, 12);
console.log(subStr);
```

**Output:** "World"

## 4. String substr(start, length):

- Extracts a specified number of characters from a string, starting at the specified index.

**Example:**

```
const str = "Hello, World!";
const subStr = str.substr(7, 5);
console.log(subStr);
```

**Output:** "World"

## 5. String replace(oldValue, newValue):

- Replaces the first occurrence of `oldValue` with `newValue`.

**Example:**

```
const str = "Hello, World!";
const newStr = str.replace("World", "Universe");
console.log(newStr);
```

**Output:** "Hello, Universe!"

## 6. String replaceAll(oldValue, newValue) (Available in ES2021 and later):

- Replaces all occurrences of `oldValue` with `newValue`.

**Example**:

```
const str = "Hello, World!";
const newStr = str.replaceAll("o", "0");
console.log(newStr);
```

**Output:** "Hell0, W0rld!"

## 7. String toUpperCase():

- Converts the string to uppercase.

**Example:**

```
const str = "Hello, World!";
const upperCaseStr = str.toUpperCase();
console.log(upperCaseStr);
```

**Output:** "HELLO, WORLD!"

## 8. String toLowerCase():

- Converts the string to lowercase.

**Example:**

```
const str = "Hello, World!";
const lowerCaseStr = str.toLowerCase();
console.log(lowerCaseStr);
```

**Output:** "hello, world!"

### 9. String concat():

- Combines one or more strings and returns a new string.

**Example:**

```
const str1 = "Hello, ";
const str2 = "World!";
const combinedStr = str1.concat(str2);
console.log(combinedStr);
```

**Output:** "Hello, World!"

## 10. String trim():

- Removes whitespace from the beginning and end of the string.

**Example:**

```
const str = "   Hello, World!   ";
const trimmedStr = str.trim();
console.log("String trim:", trimmedStr);
```

**Output:** "Hello, World!"

## 11. String trimStart() or String trimLeft():

- Removes whitespace from the beginning of the string.

**Example:**

```
const str = "   Hello, World!   ";
const trimmedStartStr = str.trimStart();
console.log("String trimStart:", trimmedStartStr);
```

**Output:** "Hello, World!   "

## 12. String trimEnd() or String trimRight():

- Removes whitespace from the end of the string.

**Example:**

```
const str = "   Hello, World!   ";
```

```
const trimmedEndStr = str.trimEnd();
console.log("String trimEnd:", trimmedEndStr);
```

**Output:** "  Hello, World!"

**13. String padStart(targetLength, padString):**

- Pads the string with a specified character (or space) until it reaches the desired length.

**Example:**

```
const str = "5";
const paddedStartStr = str.padStart(3, "0");
console.log("String padStart:", paddedStartStr);
```

**Output:** "005"

## 14. String padEnd(targetLength, padString):

- Pads the string from the end with a specified character (or space) until it reaches the desired length.

**Example:**

```
const str = "5";
const paddedEndStr = str.padEnd(3, "0");
console.log("String padEnd:", paddedEndStr);
```

**Output:** "500"

## 15. String charAt(index):

- Returns the character at the specified index.

**Example:**

```
const str = "Hello, World!";
const char = str.charAt(7);
console.log("String charAt:", char);
```

**Output:** "W"

## 16. String charCodeAt(index):

- Returns the Unicode value (integer) of the character at the specified index.

**Example:**

```
const str = "Hello, World!";
const charCode = str.charCodeAt(7);
console.log("String charCodeAt:", charCode);
```

**Output:** 87

## 17. String split(separator):

- Splits the string into an array of substrings based on the specified separator.

**Example:**

```
const str = "apple,banana,cherry";
const fruits = str.split(",");
console.log("String split:", fruits);
```

**Output:** ["apple", "banana", "cherry"]

## ❖ Over all string function in one Example:

- String length
```
const strLength = "Hello, World!".length;
console.log("String length:", strLength); // Output: 13
```
- String slice(start, end)
```
const strSlice = "Hello, World!".slice(0, 5);
console.log("String slice:", strSlice); // Output: "Hello"
```
- String substring(start, end)
```
const strSubstring = "Hello, World!".substring(7, 12);
console.log("String substring:", strSubstring); // Output: "World"
```
- String substr(start, length)
```
const strSubstr = "Hello, World!".substr(7, 5);
console.log("String substr:", strSubstr); // Output: "World"
```
- String replace(oldValue, newValue)
```
const strReplace = "Hello, World!".replace("World", "Universe");
console.log("String replace:", strReplace); // Output: "Hello, Universe!"
```
- String replaceAll(oldValue, newValue)
```
const strReplaceAll = "Hello, World!".replaceAll("o", "0");
console.log("String replaceAll:", strReplaceAll); // Output: "Hell0, W0rld!"
```
- String toUpperCase()
```
const strUpperCase = "Hello, World!".toUpperCase();
console.log("String toUpperCase:", strUpperCase); // Output: "HELLO, WORLD!"
```
- String toLowerCase()
```
const strLowerCase = "Hello, World!".toLowerCase();
console.log("String toLowerCase:", strLowerCase); // Output: "hello, world!"
```
- String concat()
```
const strConcat1 = "Hello, ";
const strConcat2 = "World!";
const concatenatedStr = strConcat1.concat(strConcat2);
console.log("String concat:", concatenatedStr); // Output: "Hello, World!"
```
- String trim()
```
const strTrim = "   Hello, World!   ".trim();
console.log("String trim:", strTrim); // Output: "Hello, World!"
```
- String trimStart() or trimLeft()
```
const strTrimStart = "   Hello, World!   ".trimStart();
console.log("String trimStart:", strTrimStart); // Output: "Hello, World!   "
```
- String trimEnd() or trimRight()
```
const strTrimEnd = "   Hello, World!   ".trimEnd();
console.log("String trimEnd:", strTrimEnd); // Output: "   Hello, World!"
```
- String padStart(targetLength, padString)
```
const strPadStart = "5".padStart(3, "0");
console.log("String padStart:", strPadStart); // Output: "005"
```
- String padEnd(targetLength, padString)
```
const strPadEnd = "5".padEnd(3, "0");
console.log("String padEnd:", strPadEnd); // Output: "500"
```
- String charAt(index)
```
const strCharAt = "Hello, World!".charAt(7);
console.log("String charAt:", strCharAt); // Output: "W"
```
- String charCodeAt(index)
```
const charCode = "Hello, World!".charCodeAt(7);
console.log("String charCodeAt:", charCode); // Output: 87
```
- String split(separator)
```
const strSplit = "apple,banana,cherry".split(",");
console.log("String split:", strSplit); // Output: ["apple", "banana",
```

**Example:**

```
let s="hello to all"
console.log(s[11])//prints the element stored at index 11
console.log(s.length)//prints the length of string
console.log(s.charAt(1))//prints the element stored at index 1
let res=s.toUpperCase()
console.log(res)
console.log(s.includes("to"))//returns true if it is present in string else false will be returned
console.log(s.startsWith("h"))//returns true if starting element is present
console.log(s.endsWith("all"))
console.log(s.replace("","$"))//it will add $ at the start
console.log(s.replace("hello","$"))
console.log(s.replace("h","$"))//replaces only the first occurence
console.log(s.replaceAll("l","-"))//replaces all the occurences
console.log(s[9])
console.log(s.charCodeAt(9))//display UNICODE value for given element
console.log(String.fromCharCode(65))
let r="*".repeat(3)//repeats the given string value 3 times
console.log(r)
console.log(s.slice(0,3))//equivalent to slicing operation
console.log(s.slice(-3))
//Negative values are not supported in substring method
console.log(s.substring(0,5))//the first value indicates initial index position & second value indicates last index position
console.log(s.substring(5))
console.log(s.substr(9,3))//the first value indicates index position & second value indicates length of substring to be displayed
console.log(s.substr(9,3))
console.log(s.split())//o/p [ 'hello to all' ]
console.log(s.split(" "))// o/p [ 'hello', 'to', 'all' ]
console.log(s.split(""))
```

**Output:**

```
 [
  'h', 'e', 'l', 'l',
  'o', ' ', 't', 'o',
  ' ', 'a', 'l', 'l'
        ]
let r=s.split(" ")
console.log(r.join(""))//o/p hellotoall
console.log(r.join(" "))//o/p hello to all
```

**Exercise:**

```
const prompt = require("prompt-sync")();
let s=prompt("Enter a string ")
for(let i=0;i<s.length;i++)
   console.log(i,s[i])
```

**//Program to print the count of each element from given string**

```
let d={}
for(let i of s){
   if(d[i]===undefined)
      d[i]=1
   else
      d[i]+=1
}
console.log(d)
```

**//Program to print the count of given element from given string**

```
let ele=prompt("Enter the element to be searched- ")
let ct=0
for(let i of s){
   if(i===ele)
      ct+=1
}
console.log(`The count of ${ele} in given array is ${ct}`)
//Program to print the reverse of string
let rev=""
for(let i of s){
   rev=i+rev
}
console.log(rev)
```

**//Program to check whether given string is palindrome or not**

```
let rev=""
for(let i of s){
   rev=i+rev
}
if (rev===s)
console.log("palindrome")
else
console.log("Not palindrome")
```

## ❖ String Search Methods:

1) String indexOf()
2) String lastIndexOf()
3) String search()
4) String match()
5) String matchAll()
6) String includes()
7) String startsWith()
8) String endsWith()

### 1. String indexOf(substring, startIndex):

- Returns the index of the first occurrence of a substring, or -1 if not found.

**Example:**

```
const str = "Hello, World!";
const index = str.indexOf("World");
console.log("String indexOf:", index);
```

**Output: 7**

### 2. String lastIndexOf(substring, startIndex):

- Returns the index of the last occurrence of a substring, or -1 if not found.

**Example:**

```
const str = "Hello, World, World!";
const lastIndex = str.lastIndexOf("World");
console.log("String lastIndexOf:", lastIndex);
```

**Output:** 13

### 3. String search(regexp):

- Searches the string for a specified pattern (regular expression) and returns the index of the first match, or -1 if not found.

**Example:**

```
const str = "The cat and the hat";
const index = str.search(/cat/);
console.log("String search:", index);
```

**Output:** 4

### 4. String match(regexp):

- Searches the string for a specified pattern (regular expression) and returns an array of the matches.

**Example:**

```
const str = "The cat and the hat";
const matches = str.match(/(cat|hat)/g);
console.log("String match:", matches);
```

**Output:** ["cat", "hat"]

## 5. String matchAll(regexp):

- Searches the string for a specified pattern (regular expression) and returns an iterable containing all matches and their capture groups.

**Example:**

```
const str = "The cat and the hat";
const matchIterator = str.matchAll(/(cat|hat)/g);
for (const match of matchIterator) {
  console.log("String matchAll:", match[0]);
}
```

**Output:**

"cat"
"hat"

## 6. String includes(substring):

- Checks if the string contains the specified substring and returns a boolean.

**Example:**

```
const str = "Hello, World!";
const includes = str.includes("World");
console.log("String includes:", includes);
```

**Output:** true

7. String startsWith(substring):

- Checks if the string starts with the specified substring and returns a boolean.

**Example:**

```
const str = "Hello, World!";
const startsWith = str.startsWith("Hello");
console.log("String startsWith:", startsWith);
```

**Output:** true

## 8. String endsWith(substring):

- Checks if the string ends with the specified substring and returns a boolean.

**Example:**

```
const str = "Hello, World!";
const endsWith = str.endsWith("World!");
console.log("String endsWith:", endsWith);
```

**Output:** true

# THIS KEYWORD

- his keyword is a special variable that is automatically created in the scope of every function.
- The this keyword is a reference variable that refers to the current object.
- Which object depends on how this is being invoked (used or called).
- The this keyword refers to different objects depending on how it is used:
- In an object method, this refers to the object.
- Alone, this refers to the global object.
- In a function, this refers to the global object.
- In a function, in strict mode, this is undefined.
- In an event, this refers to the element that received the event.
- Methods like call(), apply(), and bind() can refer this to any object.

## 1. Global Context:

- In the global context (outside of any function or object), this refers to the global object, which is window in a browser environment and global in Node.js.

**Example:**

```
console.log(this === window); // true (in a browser)
console.log(this === global); // true (in Node.js)
```

## 2. Function Context:

- In a regular function, this refers to the object that called the function. It can change depending on how the function is invoked.

**Example:**

```
function greet() {
  console.log(`Hello, ${this.name}!`);}
const person = { name: "Sangam" };
person.greet = greet;
person.greet(); // Outputs "Hello, Sangam!"
const anotherGreet = person.greet;
anotherGreet(); // Outputs "Hello, undefined!"
```

**Note:** In the first call to person.greet(), this inside the greet function refers to the person object. In the second call, when anotherGreet is called without any context, this refers to the global object (window in a browser), so this.name is undefined.

## 3. Arrow Functions:

- Arrow functions have a fixed this value. They inherit this from the surrounding lexical context, which means they use this value of the enclosing function or scope.

**Example:**

```
const person = {
  name: "Alice",
  greet: function () {
    setTimeout(() => {
      console.log(`Hello, ${this.name}!`);
    }, 1000);},};
person.greet(); // Outputs "Hello, Alice!" after a 1-second delay
```

**Note:** In this example, the arrow function inside the setTimeout callback captures this from the person object because it's defined within the greet method.

## 4. Constructor Functions:

- When a function is used as a constructor with the new keyword, this refers to the newly created object.

**Example:**

```
function Person(name) {
  this.name = name;
}
const person = new Person("raj");
console.log(person.name); // Outputs "raj"
```

**Note:** In this case, this inside the Person constructor refers to the newly created person object.

## 5. Event Handlers:

- In the context of event handlers (e.g., for DOM elements), this refers to the DOM element that triggered the event.

**Example (HTML and JavaScript):**

```
<button id="myButton">Click me</button>
<script>
  const button = document.getElementById("myButton");
  button.addEventListener("click", function () {
    console.log(this.textContent); // Outputs "Click me"
  });
</script>
```

**Note:**

- In this example, this inside the event listener function refers to the button element because it triggered the click event.

## 6. Global Object in Strict Mode:

- In strict mode ("use strict"), when a function is called without any context (not as a method or constructor), this is undefined rather than referring to the global object.

**Example:**

```
"use strict";
function strictFunction() {
  console.log(this);
}
strictFunction(); // Outputs "undefined"
```

**Note:** this in JavaScript is a dynamic keyword whose value depends on how a function is invoked.

This Precedence

- To determine which object this refers to; use the following precedence of order.

| Precedence | Object |
|---|---|
| 1 | bind() |
| 2 | apply() and call() |
| 3 | Object method |
| 4 | Global scope |

- Is this in a function being called using bind()?
- Is this in a function being called using apply()?
- Is this in a function being called using call()?
- Is this in an object function (method)?
- Is this in a function in the global scope.

# HOISTING:

- JavaScript hoisting is a behavior where variable and function declarations are moved to the top of their containing scope during the compilation phase before the code is executed.
- Hoisting is a mechanism in JavaScript that moves declaration of variables and functions at the top.

## ❖ Variable Hoisting:

**Example:**

```
console.log(myVar); // Outputs: undefined
var myVar = 10;
console.log(myVar); // Outputs: 10
```

**Example:**

```
console.log(a) //ReferenceError: Cannot access 'a' before initialization
let a=6
console.log(a) //output: 6
```

**Example:**

```
console.log(a) //ReferenceError: Cannot access 'a' before initialization
const a=6
console.log(a) // 6
```

## ❖ Function Hoisting:

- Function declarations, unlike variable declarations, are fully hoisted. This means that both the function name and its implementation are moved to the top of the scope, making the function available for use before it's declared in the code.

**Example:**

```
sayHello(); // Outputs: Hello!
function sayHello() {
  console.log("Hello!");
}
```

**Note:** here sayHello function are declered with function keyword so output come

Example:

```
myFunc(); // TypeError: myFunc is not a function
var myFunc = function () {
  console.log("Hello!");
};
```

**Note:** the myFunc variable is hoisted, but it's not initialized as a function until the assignment statement is encountered. Thus, attempting to call it before the assignment results in a TypeError.

# ARRAY:

- Array is a data structure used to store and manage collections of values.
- Arrays can hold elements of different data types, including numbers, strings, objects, functions, and other arrays.
- arrays are indexed, ordered, and mutable, meaning you can add, remove, and modify elements.

## 1. Creating Arrays:

- You can create an array in JavaScript using either literal notation or the `Array` constructor:

```
// Literal notation
const fruits = ["apple", "banana", "cherry"];
// Using the Array constructor
const cars = new Array("Toyota", "Honda", "Ford");
```

## 2. Accessing Elements:

- You can access array elements using square brackets and index, with the first element at index 0:

```
const fruits = ["apple", "banana", "cherry"];
console.log(fruits[0]); // "apple"
console.log(fruits[1]); // "banana"
```

## 3. Modifying Arrays:

- Arrays in JavaScript are mutable, which means you can change their content by adding, removing, or updating elements:

```
const fruits = ["apple", "banana", "cherry"];
// Adding elements
fruits.push("orange"); // Adds "orange" to the end
fruits.unshift("grape"); // Adds "grape" to the beginning
// Removing elements
fruits.pop(); // Removes the last element ("orange")
fruits.shift(); // Removes the first element ("grape")
// Updating elements
fruits[1] = "strawberry"; // Changes "banana" to "strawberry"
```

## 4. Array Length:

- You can find the number of elements in an array using the `length` property:

```
const fruits = ["apple", "banana", "cherry"];
console.log(fruits.length); // 3
```

## 5. Iterating Through Arrays:

- You can loop through the elements of an array using `for` loops, `forEach`, `for...of`, or other looping constructs:

```
const fruits = ["apple", "banana", "cherry"];
// Using a for loop
for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
}
// Using forEach
fruits.forEach(function (fruit) {
  console.log(fruit);
});
// Using for...of
```

```
for (const fruit of fruits) {
  console.log(fruit);
}
```

## 6. Array Methods:

- JavaScript provides a rich set of array methods for performing common operations, such as filtering, mapping, reducing, and sorting:

```
const numbers = [1, 2, 3, 4, 5];
// Filtering
const evenNumbers = numbers.filter(function (number) {
  return number % 2 === 0;
});
// Mapping
const doubledNumbers = numbers.map(function (number) {
  return number * 2;
});
// Reducing
const sum = numbers.reduce(function (accumulator, currentValue) {
  return accumulator + currentValue;
}, 0);
// Sorting
const sortedNumbers = numbers.sort(function (a, b) {
  return a - b;
});
```

## 7. Adding and Removing Elements:

- You can add and remove elements from the beginning, end, or specified positions in an array:

```
const fruits = ["apple", "banana", "cherry"];
// Adding elements at the end
fruits.push("orange");
// Removing elements from the end
fruits.pop();
// Adding elements at the beginning
fruits.unshift("grape");
// Removing elements from the beginning
fruits.shift();
// Adding elements at a specified position
fruits.splice(1, 0, "kiwi");
// Removing elements from a specified position
fruits.splice(2, 1);
```

## 8. Searching and Finding Elements:

- You can search for elements in an array using methods like `indexOf`, `lastIndexOf`, `find`, and `findIndex`:

```
const fruits = ["apple", "banana", "cherry", "banana"];
console.log(fruits.indexOf("cherry")); // 2
console.log(fruits.lastIndexOf("banana")); // 3
const foundFruit = fruits.find(function (fruit) {
  return fruit.length > 5;
});
```

```
const foundIndex = fruits.findIndex(function (fruit) {
  return fruit.length > 5;
});
```

### 9. Concatenating Arrays:

- You can concatenate arrays using the `concat` method or the spread operator (`...`):

```
const fruits1 = ["apple", "banana"];
const fruits2 = ["cherry", "kiwi"];
const mergedFruits = fruits1.concat(fruits2);
const spreadFruits = [...fruits1, ...fruits2];
```

### 10. Multidimensional Arrays:

- JavaScript allows you to create arrays of arrays, forming multidimensional arrays:

```
const matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9],
];
console.log(matrix[0][1]); // 2
```

### 11. Common Array Mistakes:

- Forgetting to use square brackets when accessing array elements (e.g., `array.0` is incorrect; use `array[0]`).
- Using non-integer values as array indices (e.g., `array["key"]` is incorrect).
- Using the `==` operator for array comparisons (use `===` for strict equality).

## ❖ array function:

1) push(): Adds elements to the end of an array.
2) pop(): Removes the last element from an array.
3) unshift(): Adds elements to the beginning of an array.
4) shift(): Removes the first element from an array.
5) concat(): Combines two or more arrays into a new array.
6) slice(): Creates a shallow copy of a portion of an array.
7) splice(): Modifies an array by adding, removing, or replacing elements.
8) forEach(): Executes a function for each element in an array.
9) map(): Creates a new array by applying a function to each element.
10) filter(): Creates a new array with elements that pass a test.
11) reduce(): Reduces an array to a single value using a function.
12) find(): Returns the first element that satisfies a condition.
13) indexOf(): Returns the index of the first occurrence of an element.
14) lastIndexOf(): Returns the index of the last occurrence of an element.
15) some(): Checks if at least one element satisfies a condition.
16) every(): Checks if all elements satisfy a condition.
17) sort(): Sorts elements in an array in place.
18) reverse(): Reverses the order of elements in an array.
19) join(): Combines elements into a string with a specified separator.
20) includes(): Checks if an array contains a specific element.

## Example:

**1. push()`:** Adds one or more elements to the end of an array and returns the new length of the array.

```
const fruits = ["apple", "banana"];
fruits.push("cherry"); // ["apple", "banana", "cherry"]
```

**2. pop()`:** Removes the last element from an array and returns that element.

```
const fruits = ["apple", "banana", "cherry"];
const removed = fruits.pop(); // "cherry"
```

**3. `unshift()`:** Adds one or more elements to the beginning of an array and returns the new length of the array.

```
const fruits = ["banana", "cherry"];
fruits.unshift("apple"); // ["apple", "banana", "cherry"]
```

**4. shift()`:** Removes the first element from an array and returns that element.

```
const fruits = ["apple", "banana", "cherry"];
const removed = fruits.shift(); // "apple"
```

**5. `concat()`:** Combines two or more arrays and returns a new array.

```
const fruits1 = ["apple", "banana"];
const fruits2 = ["cherry", "kiwi"];
const mergedFruits = fruits1.concat(fruits2); // ["apple", "banana", "cherry", "kiwi"]
```

**6. `slice()`:** Returns a shallow copy of a portion of an array into a new array.

```
const fruits = ["apple", "banana", "cherry", "kiwi"];
const slicedFruits = fruits.slice(1, 3); // ["banana", "cherry"]
```

**7. splice()`:** Changes the contents of an array by removing, replacing, or adding elements in place.

```
const fruits = ["apple", "banana", "cherry"];
fruits.splice(1, 1, "kiwi"); // ["apple", "kiwi", "cherry"]
```

**8. `forEach()`:** Executes a provided function once for each array element.

```
const numbers = [1, 2, 3];
numbers.forEach(function (number) {
  console.log(number);
});
```

**9. `map()`:** Creates a new array with the results of calling a provided function on every element in the array.

```
const numbers = [1, 2, 3];
const doubledNumbers = numbers.map(function (number) {
  return number * 2;
});
```

**10. filter()`:** Creates a new array with all elements that pass a test provided by a given function.

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(function (number) {
  return number % 2 === 0;
});
```

**11. reduce()`:** Applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce(function (accumulator, currentValue) {
  return accumulator + currentValue;
}, 0);
```

**12. find()`:** Returns the first element in the array that satisfies the provided testing function.

```
const numbers = [1, 2, 3, 4, 5];
const found = numbers.find(function (number) {
  return number > 2;
});
```

**13. `indexOf()`:** Returns the first index at which a given element can be found in the array, or -1 if it's not present.

```
const fruits = ["apple", "banana", "cherry"];
const index = fruits.indexOf("banana"); // 1
```

**14. lastIndexOf()`:** Returns the last index at which a given element can be found in the array, or -1 if it's not present.

```
const fruits = ["apple", "banana", "cherry", "banana"];
const lastIndex = fruits.lastIndexOf("banana"); // 3
```

**15. `some()`:** Checks if at least one element in the array satisfies the provided testing function.

```
const numbers = [1, 2, 3, 4, 5];
const hasEvenNumber = numbers.some(function (number) {
  return number % 2 === 0;
});
```

**16. every()`:** Checks if all elements in the array satisfy the provided testing function.

```
const numbers = [2, 4, 6, 8, 10];
const allEven = numbers.every(function (number) {
  return number % 2 === 0;
});
```

**17. `sort()`:** Sorts the elements of an array in place and returns the sorted array.

```
const fruits = ["cherry", "apple", "banana"];
fruits.sort(); // ["apple", "banana", "cherry"]
```

**18. reverse()`:** Reverses the order of elements in an array in place.

```
const numbers = [1, 2, 3, 4, 5];
numbers.reverse(); // [5, 4, 3, 2, 1]
```

**19. `join()`:**Combines all elements of an array into a single string and returns it.

```
const fruits = ["apple", "banana", "cherry"];
const joinedFruits = fruits.join(", "); // "apple, banana, cherry"
```

**20. includes()`:** Checks if an array includes a specific element and returns `true` or `false`.

```
const fruits = ["apple", "banana", "cherry"];
const includesBanana = fruits.includes("banana"); // true
```

# DATE OBJECT

- Date object is used to work with dates and times. It allows you to represent and manipulate dates, perform date arithmetic, and format dates for display. The Date object is a core part of JavaScript and is commonly used in various web applications.

## ❖ Date Object Methods:

- The Date object provides various methods for working with dates and times:

**1)** getDate(), getMonth(), getFullYear(): Get the day, month (0-11), and year components of a date.
**2)** getHours(), getMinutes(), getSeconds(), getMilliseconds(): Get the time components of a date.
**3)** getDay(): Get the day of the week (0 = Sunday, 1 = Monday, ...).
**4)** toString(), toDateString(), toTimeString(): Convert a date to a string.
**5)** toLocaleString(), toLocaleDateString(), toLocaleTimeString(): Convert a date to a localized string.
**6)** getTime(): Get the timestamp (milliseconds since January 1, 1970).
**7)** setDate(), setMonth(), setFullYear(): Set the day, month, and year components of a date.
**8)** setHours(), setMinutes(), setSeconds(), setMilliseconds(): Set the time components of a date.
**9)** setTime(): Set the timestamp.

## Example of how to use the getDate(), getMonth(), and getFullYear():

```
// Create a Date object for 30 September 2023
const date = new Date(2023,9,30); // Month is 0-based (0 = January, 1 = February, ...), so 8 represents September
// Get the day, month, and year components
const day = date.getDate();
const month = date.getMonth(); // Returns 8 (0-based, so 8 is September)
const year = date.getFullYear();// Output the components
console.log(`Day: ${day}`);     // Output: Day: 30
console.log(`Month: ${month}`); // Output: Month: 9
console.log(`Year: ${year}`);   // Output: Year: 2023
```

## output:

```
Day: 30
Month: 9
Year: 2023
```

## ❖ How to display current date:

```
// Create a Date object for the current date
const currentDate = new Date();
// Get the day, month, and year components
const day = currentDate.getDate();
const month = currentDate.getMonth() + 1; // Month is 0-based, so add 1 to get the actual month
const year = currentDate.getFullYear();
// Format the components as a date string
const formattedDate = `${year}-${month.toString().padStart(2, '0')}-${day.toString().padStart(2, '0')}`;
// Output the current date
console.log(`Current Date: ${formattedDate}`);
```

**Output:**

urrent Date: 2023-09-30

**Example of how to get the local date and time using the getHours(), getMinutes(), getSeconds(), and getMilliseconds():**

```
// Create a Date object for the current local date and time
const localDate = new Date();
console.log(localDate)
// Get the local time components
const hours = localDate.getHours();
const minutes = localDate.getMinutes();
const seconds = localDate.getSeconds();
const milliseconds = localDate.getMilliseconds();// Output the local time components
console.log(`Local Hours: ${hours}`);
console.log(`Local Minutes: ${minutes}`);
console.log(`Local Seconds: ${seconds}`);
console.log(`Local Milliseconds: ${milliseconds}`);
```

**output:**

```
2023-09-20T07:32:29.621Z
Local Hours: 13
Local Minutes: 2
Local Seconds: 29
Local Milliseconds: 621
```

**Example:**

```
// Create a Date object for a specific date and time
const date = new Date("2023-09-15T14:30:45.123Z"); // September 15, 2023, 14:30:45.123
UTC
// Get the day of the week (0 = Sunday, 1 = Monday, ...)
const dayOfWeek = date.getDay();
// Convert the date to string formats
const dateString = date.toString();
const dateOnlyString = date.toDateString();
const timeOnlyString = date.toTimeString();
// Convert the date to localized string formats
const localizedString = date.toLocaleString();
const localizedDateString = date.toLocaleDateString();
const localizedTimeString = date.toLocaleTimeString();
// Get the timestamp (milliseconds since January 1, 1970)
const timestamp = date.getTime();
// Set the day, month, and year components of the date
date.setDate(10);    // Set the day to 10
date.setMonth(2);    // Set the month to March (0-based, so 2 represents March)
date.setFullYear(2022); // Set the year to 2022
// Set the time components of the date
date.setHours(18);        // Set the hours to 18 (6 PM)
date.setMinutes(45);      // Set the minutes to 45
date.setSeconds(30);      // Set the seconds to 30
date.setMilliseconds(500); // Set the milliseconds to 500
```

```
// Set the date using a timestamp (milliseconds since January 1, 1970)
date.setTime(1678554330500); // Set the date and time to a specific timestamp// Output the results
console.log(`Day of the Week: ${dayOfWeek}`);
console.log(`Date as String: ${dateString}`);
console.log(`Date Only String: ${dateOnlyString}`);
console.log(`Time Only String: ${timeOnlyString}`);
console.log(`Localized Date String: ${localizedString}`);
console.log(`Localized Date Only String: ${localizedDateString}`);
console.log(`Localized Time Only String: ${localizedTimeString}`);
console.log(`Timestamp: ${timestamp}`);
console.log(`Modified Date: ${date.toString()}`);
```

**output:**

```
Day of the Week: 5
Date as String: Fri Sep 15 2023 20:00:45 GMT+0530 (India Standard Time)
Date Only String: Fri Sep 15 2023
Time Only String: 20:00:45 GMT+0530 (India Standard Time)
Localized Date String: 15/9/2023, 8:00:45 pm
Localized Date Only String: 15/9/2023
Localized Time Only String: 8:00:45 pm
Timestamp: 1694788245123
Modified Date: Sat Mar 11 2023 22:35:30 GMT+0530 (India Standard Time)
```

# MATH IN JAVASCRIPT

- The JavaScript math object provides several constants and methods to perform mathematical operation.
  1) Math.abs(x): Returns the absolute value of a number x.
  2) Math.acos(x): Returns the arccosine (in radians) of a number x, where x is in the range [-1, 1].
  3) Math.asin(x): Returns the arcsine (in radians) of a number x, where x is in the range [-1, 1].
  4) Math.atan(x): Returns the arctangent (in radians) of a number x.
  5) Math.cbrt(x): Returns the cube root of a number x.
  6) Math.ceil(x): Returns the smallest integer greater than or equal to a number x.
  7) Math.cos(x): Returns the cosine of a number x (assumed to be in radians).
  8) Math.cosh(x): Returns the hyperbolic cosine of a number x.
  9) Math.exp(x): Returns the exponential value of a number x.
  10) Math.floor(x): Returns the largest integer less than or equal to a number x.
  11) Math.hypot(...args): Returns the square root of the sum of squares of its arguments, effectively calculating the hypotenuse of a right triangle.
  12) Math.log(x): Returns the natural logarithm (base e) of a number x.
  13) Math.max(...args): Returns the maximum value among a list of numbers passed as arguments.
  14) Math.min(...args): Returns the minimum value among a list of numbers passed as arguments.
  15) Math.pow(x, y): Returns x raised to the power of y.
  16) Math.random(): Returns a random number between 0 (inclusive) and 1 (exclusive).
  17) Math.round(x): Returns the value of a number x rounded to the nearest integer.
  18) Math.sign(x): Returns the sign of a number x (1 for positive, -1 for negative, 0 for zero, and NaN for NaN).
  19) Math.sin(x): Returns the sine of a number x (assumed to be in radians).
  20) Math.sinh(x): Returns the hyperbolic sine of a number x.
  21) Math.sqrt(x): Returns the square root of a number x.
  22) Math.tan(x): Returns the tangent of a number x (assumed to be in radians).
  23) Math.tanh(x): Returns the hyperbolic tangent of a number x.
  24) Math.trunc(x): Returns the integer part of a number x (removing the decimal part).

**Example:**

```
// Math.abs(x)
const absValue = Math.abs(-4.5);
console.log(`Math.abs(-4.5) => ${absValue}`); // Output: 4.5
// Math.acos(x)
const acosValue = Math.acos(0.5);
console.log(`Math.acos(0.5) => ${acosValue} radians`); // Output: 1.0471975511965979 radians
// Math.asin(x)
const asinValue = Math.asin(0.5);
console.log(`Math.asin(0.5) => ${asinValue} radians`); // Output: 0.5235987755982989 radians
// Math.atan(x)
const atanValue = Math.atan(1);
console.log(`Math.atan(1) => ${atanValue} radians`); // Output: 0.7853981633974483 radians
// Math.cbrt(x)
```

```
const cbrtValue = Math.cbrt(8);
console.log(`Math.cbrt(8) => ${cbrtValue}`); // Output: 2
// Math.ceil(x)
const ceilValue = Math.ceil(4.2);
console.log(`Math.ceil(4.2) => ${ceilValue}`); // Output: 5
// Math.cos(x)
const cosValue = Math.cos(Math.PI);
console.log(`Math.cos(Math.PI) => ${cosValue}`); // Output: -1
// Math.cosh(x)
const coshValue = Math.cosh(1);
console.log(`Math.cosh(1) => ${coshValue}`); // Output: 1.5430806348152437
// Math.exp(x)
const expValue = Math.exp(1);
console.log(`Math.exp(1) => ${expValue}`); // Output: 2.718281828459045
// Math.floor(x)
const floorValue = Math.floor(4.8);
console.log(`Math.floor(4.8) => ${floorValue}`); // Output: 4
// Math.hypot(...args)
const hypotValue = Math.hypot(3, 4);
console.log(`Math.hypot(3, 4) => ${hypotValue}`); // Output: 5
// Math.log(x)
const logValue = Math.log(Math.E);
console.log(`Math.log(Math.E) => ${logValue}`); // Output: 1
// Math.max(...args)
const maxValue = Math.max(3, 8, 1, 6, 4);
console.log(`Math.max(3, 8, 1, 6, 4) => ${maxValue}`); // Output: 8
// Math.min(...args)
const minValue = Math.min(3, 8, 1, 6, 4);
console.log(`Math.min(3, 8, 1, 6, 4) => ${minValue}`); // Output: 1
// Math.pow(x, y)
const powValue = Math.pow(2, 3);
console.log(`Math.pow(2, 3) => ${powValue}`); // Output: 8
// Math.random()
const randomValue = Math.random();
console.log(`Math.random() => ${randomValue}`); // Output: A random number between 0
(inclusive) and 1 (exclusive)
// Math.round(x)
const roundValue = Math.round(4.6);
console.log(`Math.round(4.6) => ${roundValue}`); // Output: 5
// Math.sign(x)
const signValue = Math.sign(-7);
console.log(`Math.sign(-7) => ${signValue}`); // Output: -1
// Math.sin(x)
const sinValue = Math.sin(Math.PI / 2);
console.log(`Math.sin(Math.PI / 2) => ${sinValue}`); // Output: 1
// Math.sinh(x)
const sinhValue = Math.sinh(1);
console.log(`Math.sinh(1) => ${sinhValue}`); // Output: 1.1752011936438014
```

```
// Math.sqrt(x)
const sqrtValue = Math.sqrt(9);
console.log(`Math.sqrt(9) => ${sqrtValue}`); // Output: 3
// Math.tan(x)
const tanValue = Math.tan(Math.PI / 4);
console.log(`Math.tan(Math.PI / 4) => ${tanValue}`); // Output: 1
// Math.tanh(x)
const tanhValue = Math.tanh(1);
console.log(`Math.tanh(1) => ${tanhValue}`); // Output: 0.7615941559557649
// Math.trunc(x)
const truncValue = Math.trunc(4.9);
console.log(`Math.trunc(4.9) => ${truncValue}`); // Output: 4
```

# NUMBER OBJECT

- The JavaScript number object enables you to represent a numeric value.

1) Number.isNaN(value): Checks if a value is NaN.
2) Number.isFinite(value): Checks if a value is finite (not NaN, Infinity, or -Infinity).
3) Number.parseInt(string, radix): Parses a string and returns an integer.
4) Number.parseFloat(string): Parses a string and returns a floating-point number.
5) Number.toFixed(digits): Converts a number to a string with a fixed number of decimal places.
6) Number.toPrecision(precision): Converts a number to a string with a specified number of significant digits.
7) Number.toString(base): Converts a number to a string in a specified base.
8) Number.isInteger(value): Checks if a value is an integer.
9) Number.MAX_VALUE: Represents the maximum numeric value.
10) Number.MIN_VALUE: Represents the smallest positive numeric value.
11) Number.POSITIVE_INFINITY: Represents positive infinity.
12) Number.NEGATIVE_INFINITY: Represents negative infinity.

**Example:**

```
// Example of JavaScript Number Methods
// Number.isNaN(value)
const isNaNResult = Number.isNaN(5 / 'abc');
console.log(`Number.isNaN(5 / 'abc') => ${isNaNResult}`); // Output: true
// Number.isFinite(value)
const isFiniteResult = Number.isFinite(42);
console.log(`Number.isFinite(42) => ${isFiniteResult}`); // Output: true
// Number.parseInt(string, radix)
const parseIntResult = Number.parseInt('123', 10);
console.log(`Number.parseInt('123', 10) => ${parseIntResult}`); // Output: 123
// Number.parseFloat(string)
const parseFloatResult = Number.parseFloat('3.14');
console.log(`Number.parseFloat('3.14') => ${parseFloatResult}`); // Output: 3.14
// Number.toFixed(digits)
const numToFixed = 3.14159265359;
const toFixedResult = numToFixed.toFixed(2);
console.log(`3.14159265359.toFixed(2) => ${toFixedResult}`); // Output: "3.14"
// Number.toPrecision(precision)
const numToPrecision = 12345.6789;
const toPrecisionResult = numToPrecision.toPrecision(6);
console.log(`12345.6789.toPrecision(6) => ${toPrecisionResult}`); // Output: "12345.7"
// Number.toString(base)
const numToString = 42;
const binaryStr = numToString.toString(2);
console.log(`42.toString(2) => "${binaryStr}"`); // Output: "101010"
// Number.isInteger(value)
const isIntegerResult = Number.isInteger(42);
console.log(`Number.isInteger(42) => ${isIntegerResult}`); // Output: true
// Number.MAX_VALUE
```

```
const maxValue = Number.MAX_VALUE;
console.log(`Number.MAX_VALUE => ${maxValue}`); // Output: Largest positive number
// Number.MIN_VALUE
const minValue = Number.MIN_VALUE;
console.log(`Number.MIN_VALUE => ${minValue}`); // Output: Smallest positive number greater than 0
// Number.POSITIVE_INFINITY
const positiveInfinity = Number.POSITIVE_INFINITY;
console.log(`Number.POSITIVE_INFINITY => ${positiveInfinity}`); // Output: Positive infinity
// Number.NEGATIVE_INFINITY
const negativeInfinity = Number.NEGATIVE_INFINITY;
console.log(`Number.NEGATIVE_INFINITY => ${negativeInfinity}`); // Output: Negative infinity
```

# BOOLEAN:

- JavaScript Boolean is an object that represents value in two states: true or false.

- JavaScript Boolean Properties

| Property | Description |
|---|---|
| constructor | returns the reference of Boolean function that created Boolean object. |
| prototype | enables you to add properties and methods in Boolean prototype. |

❖ **Boolean Methods:**

- Method Description

| | |
|---|---|
| toSource() | returns the source of Boolean object as a string. |
| toString() | converts Boolean into String. |
| valueOf() | converts other type into Boolean. |

# OOPS

# Object Oriented Programming

- Object:- An Object is a unique entity that contains properties and methods.
  - For example "a car" is a real-life Object, which has some characteristics like color, type, model, and horsepower and performs certain actions like driving etc.
- An Object is an instance of a class
- The object can be created in two ways in JavaScript:
  1. Object Literal
  2. Object Constructor

## 1. Object Literal:

- camelCase Naming convention followed here

**Example:**

```
let person={
   name:"Sangam Kumar",
   age:28,
   contact:900500506,
   getDetails: function(){
      console.log(`Name is ${this.name}\nAge is ${this.age}\nContact is- ${this.contact}`)
   }
}
console.log(person)
person.getDetails()
```

**Output:**

```
{
   name: 'Sangam Kumar',
   age: 28,
   contact: 900500506,
   getDetails: [Function: getDetails]
 }
 Name is Sangam Kumar
 Age is 28
 Contact is- 900500506
```

## 2. Object Constructor:

- PascalCase Naming convention followed here.
- Here Person behaves like class

**Example:**

```
function Person(name,age,address){
   this.name=name
   this.age=age
   this.address=address
}
let p1 = new Person("Jai Kumar",30,"Patna")
console.log("Name:", p1.name,'\n',"Age:",p1.age,"\n Address:",p1.address)
```

**output:**

```
Name: Jai Kumar
Age: 30
Address: Patna
```

## ❖ Declaring class in ES6:

**Example:**

```
class Person{
  constructor(name,age,address){
    this.name=name
    this.age=age
    this.address=address
  }
  getDetails(){
    console.log(`Name is ${this.name}\nAge is ${this.age}\nContact is- ${this.address}`)
  }
}
//here Person has got 3 attribute- name,age,address & one method- getDetails()
let ob1 = new Person("Sangam Kumar",29,"Bihata Patna")//here object is created using new
keyword. hence ob1 is object & Person is class
ob1.getDetails()
```

**output:**

```
Name is Sangam Kumar
Age is 29
Contact is- Bihata Patna
```

Create a user choice based Calculator using OOP

**Example:**

```
class Calculator{
  constructor(num1,num2){
    this.num1=num1
    this.num2=num2
  }
  add(){
    return this.num1+this.num2
  }
  sub(){
    return this.num1-this.num2
  }
  mul(){
    return this.num1*this.num2
  }
  div(){
    return this.num1/this.num2
  }
}
//DYNAMIC INPUT IN node js
//Install the module prompt-sync
// npm i prompt-sync
```

```
const prompt = require("prompt-sync")();
let n1=parseInt(prompt("Enter number1- "))
let n2=parseInt(prompt("Enter number2- "))
let cl=new Calculator(n1,n2)
console.log("Choose 1 for Addition\nChoose 2 for Subtraction\nChoose 3 for Multiplication\nChoose 4 for Division")
let ch=parseInt(prompt("Enter your choice: "))
if(ch===1){
  console.log( "Sum=", cl.add() )
}
else if(ch===2){
   console.log("Substration=", cl.sub() )
 }
else if(ch===3){
   console.log("Multiplication=", cl.mul() )
 }
else if(ch===4){
   console.log("Division=", cl.div() )
 }
else{
   console.log("Invalid choice")
}
```

**output:**

```
Enter number1- 10
Enter number2- 20
Choose 1 for Addition
Choose 2 for Subtraction
Choose 3 for Multiplication
Choose 4 for Division
Enter your choice: 3
Multiplication= 200
```

## ❖ Inheritance:

- Note- Multiple Inheritance is not supported in JS
- Single Inheritance

**Example:**

```
class Father{//Parent class
  constructor(fname){
    this.fname=fname
  }
  print(){
    console.log("Father name is ",this.fname)
  }
}
class Son extends Father{//Son is child class
  constructor(fname,sname){
     // super keyword for calling the above
```

```
        // class constructor
        super(fname)
        this.sname=sname
    }
    print(){
        console.log("Father name is ",this.fname)
        console.log("Child name is:",this.sname)
    }
}
ob = new Son("Sangam Kumar","rajvardhan Kumar")
ob.print()
```

**output:**

```
Father name is  Sangam Kumar
Child name is: rajvardhan Kumar
```

## ❖ Multilevel Inheritance:

**Example:**

```
class GrandFather//Grandfather is base class
{
    constructor(gname){
        this.gname=gname
    }
    disp(){
        console.log("GrandFather name is ",this.gname)
    }

}
class Father extends GrandFather//Father is child class for Grandfather

{
    constructor(gname,fname){
        super(gname)
        this.fname=fname
    }
    disp(){
        console.log("GrandFather name:",this.gname)
        console.log("Father name:",this.fname)
    }

}
class Son extends Father//Son is the child class for Father
{
    constructor(gname,fname,sname){
        super(gname,fname)
        this.sname=sname
    }
    disp(){
        console.log("GrandFather name: ",this.gname)
```

```
        console.log("Father name: ",this.fname)
        console.log("Son's name:",this.sname)

      }

    }
    ob=new Son("Sangam Kumar","Rajvardhan Kumar","Rajshekhar Kumar")
    ob.disp()
    ob1=new Father("Kumar","Rajvardhan Kumar")
    ob1.disp()
```

**output:**

```
GrandFather name:  Sangam Kumar
Father name:  Rajvardhan Kumar
Son's name: Rajshekhar Kumar
GrandFather name: Kumar
Father name: Rajvardhan Kumar
```

## ❖ Polymorphism:

**1.Method overriding---inheritance is mandatory**

**Example:**

```
class Animal{
  makeSound(){
    console.log("Animal sound")
  }}
class Cat extends Animal{
  makeSound(){
    console.log("Cat Sound-Meow")
  }}
ob=new Cat()
ob.makeSound()//makeSound method of base class is overriden in child class
```

**output:**

```
Cat Sound-Meow
```

## 2.Function overloading:

**Example:**

```
function area(shape){
  if(shape.type === "circle")
    return Math.PI * shape.radius ** 2
  else if(shape.type === "rectangle")
    return shape.width * shape.height
}
let cir = {type:"circle",radius:10}
let rec = {type:"rectangle",width:5,height:10}
console.log(area(cir))
console.log(area(rec))
```

**output:**

```
314.1592653589793
50
```

## 3. Operator Overloading:

**Example:**

```
class twksaa{
   constructor(x,y){
      this.x=x
      this.y=y
   }
   add(other){
      return new twksaa(this.x+other.x, this.y+other.y)
   }}
let v1=new twksaa(2,5)
let v2=new twksaa(10,20)
console.log(v1.add(v2))
```

**output:**

```
twksaa { x: 12, y: 25 }
```

## 4.Duck Typing:

**Example:**

```
function info(animal){
   console.log("This animal makes a sound",animal.sound)
}
let cat = {sound:"Meow"}
let dog = {sound:"bark"}
info(cat)// o/p- Meow
info(dog)// o/p- bark
```

**output:**

```
his animal makes a sound Meow
This animal makes a sound bark
```

## ❖ Abstraction- It is the process to hide unnecessary information from user

**Example:**

```
class Shape{
   findArea()//abstract method
   {
      throw new Error("This method has to be implemented in child class")
   }}
```

- Abstract class can't be instantiated
- It ensures that all other class which inherits it should override the abstract method

**Example:**

```
class Circle extends Shape{
   findArea(){
      let r=10
      console.log("Area of circle",Math.PI*r**2) } }
class Rectangle extends Shape{
   findArea(){
      let len=10
      let wid=5
      console.log("Area of Rectangle",len*wid)
```

```
    }}
ob1 = new Circle()
ob1.findArea()
ob2 = new Rectangle()
ob2.findArea()
```

## ❖ Abstraction:

**example:**

```
function Person(fname, lname) {
        let firstname = fname;
        let lastname = lname;

        let getDetails_noaccess = function () {
                return (`First name is: ${firstname} Last
                        name is: ${lastname}`);
        }

        this.getDetails_access = function () {
                return (`First name is: ${firstname}, Last
                        name is: ${lastname}`);
        }}
let person1 = new Person('Mukul', 'Latiyan');
console.log(person1.firstname);
console.log(person1.getDetails_noaccess);
console.log(person1.getDetails_access());
```

## ❖ Encapsulation:

**example:**

```
class person {
        constructor(name, id) {
                this.name = name;
                this.id = id;
        }
        add_Address(add) {
                this.add = add;
        }
        getDetails() {
                console.log(`Name is ${this.name},Address is: ${this.add}`);
        }}
let person1 = new person('ASHOK', 21);
person1.add_Address('Hyderabad');
person1.getDetails();
```

**Note-** Encapsulation refers to the hiding of data or data Abstraction which means representing essential features hiding the background detail.

Most of the OOP languages provide access modifiers to restrict the scope of a variable, but there are no such access modifiers in JavaScript, there are certain ways by which we can restrict the scope of variables within the Class/Object.

# BROWSER OBJECT MODEL(BOM):

- 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.

**Example:**

```
<!DOCTYPE html>
<html>
<body>
<script>
//window.alert("Welcome to twksaa skills cenete")
//is same as:
alert("Welcome to twksaa skills cenete")
</script>
</body>
</html>
```

**output:**

Welcome to twksaa skills cenete

- You can use a lot of properties (other objects) defined underneath the window object like document, history, screen, navigator, location, innerHeight, innerWidth,

## ❖ 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.
- 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.

## ❖ alert():

**Example:**

```
<!DOCTYPE html>
<html>
<body>
  <script type="text/javascript">
    function msg(){
     alert("Welcome!!!");
    }
    </script>
    <input type="button" value="Click Here" onclick="msg()"/>
```

```
</body>
</html>
```

## ❖ confirm():

**Example:**

```
<!DOCTYPE html>
<html>
<body>
  <script type="text/javascript">
    function msg(){
    let v= confirm("Are u sure?");
    if(v==true){
    alert("Your record deleted!");
    }
    else{
    alert("Cancel!!!");
    }
    }
    </script>
    <input type="button" value="delete record" onclick="msg()"/>
</body>
</html>
```

## ❖ prompt():

**Example:**

```
<!DOCTYPE html>
<html>
<body>
  <script type="text/javascript">
    function msg(){
    let v= prompt("Who are you?");
    alert("I am "+v);
    }
    </script>
    <input type="button" value="Click Here" onclick="msg()"/>
  </body>
</html>
```

## ❖ open():

**Example:**

```
<!DOCTYPE html>
<html>
<body>
  <script type="text/javascript">
    function msg(){
    open("http://www.twksaa.org");
    }
    </script>
    <input type="button" value="Open website" onclick="msg()"/>
  </body>
```

```
</html>
```

## ❖ close():

**Example:**

```
<!DOCTYPE html>
<html>
<body>
  <script type="text/javascript">
    function msg(){
    setTimeout(
    function(){
    alert("Welcome to Twksaa skills center after 3 seconds")
    },3000);
    }
    </script>
    <input type="button" value="Click Here" onclick="msg()"/>
  </body>
</html>
```

## ❖ History Object:

- The JavaScript History object is a built-in object that represents the browser's session history. It provides methods and properties to navigate through the user's browsing history, such as going back and forward between visited pages and manipulating the browser's history stack.

1. **back():** This method moves the browser back one page in the session history. It is equivalent to clicking the "Back" button in the browser.

```
window.history.back();
```

2. **forward():** This method moves the browser forward one page in the session history. It is equivalent to clicking the "Forward" button in the browser.
3. **go([delta]):** The `go()` method allows you to navigate through the history by a specified number of steps. A positive `delta` value moves forward, and a negative value moves backward.

```
window.history.go(2); // Move forward two pages
window.history.go(-1); // Move backward one page
```

## ❖ Properties:

1. **length:** The `length` property represents the number of entries in the session history, which is the number of pages in the browser's history stack.

```
const historyLength = window.history.length;
```

2. **state:** The `state` property allows you to associate a custom state object with a history entry. This state object can be accessed when the user navigates to that entry using the `popstate` event.

```
window.history.pushState({ data: 'custom state' }, 'Custom Title', '/new-page');
```

- Here, `{ data: 'custom state' }` is the custom state object associated with the history entry.

3. **pushState():** The `pushState()` method allows you to add a new state to the session history. This doesn't trigger a page load but instead changes the URL and adds an entry to the history stack.

```
window.history.pushState({ data: 'custom state' }, 'Custom Title', '/new-page');
```

4. **replaceState():** The `replaceState()` method is similar to `pushState()` but replaces the current state in the history stack with a new one, without adding a new entry.

**Example:**

```
history.back();//for previous page
history.forward();//for next page
history.go(2);//for next 2nd page
history.go(-2);//for previous 2nd page
Example:
<!DOCTYPE html>
<html>
<head>
 <title>JavaScript History Object Example</title>
</head>
<body>
 <h1>History Object Example</h1>
 <button onclick="goBack()">Go Back</button>
 <button onclick="goForward()">Go Forward</button>
 <button onclick="addState()">Add State</button>
 <script>
  // Function to navigate back in history
  function goBack() {
   window.history.back();
  }
  // Function to navigate forward in history
  function goForward() {
   window.history.forward();
  }
  // Function to add a new state to the history
  function addState() {
   const newState = {
    data: 'Custom State Data'
   };
   const newTitle = 'New Page Title';
   const newUrl = '/new-page';
   window.history.pushState(newState, newTitle, newUrl);
   // Display the updated history length
   alert(`History length: ${window.history.length}`);
  }
 </script>
</body>
</html>
```

# NAVIGATOR` OBJECT:

- The JavaScript `Navigator` object is a built-in object in web browsers that provides information about the client's web browser and operating system. Developers can use the `Navigator` object to access various properties and methods to determine and control the user's browsing environment. Here are some of the key properties and methods of the `Navigator` object:
  1. **navigator.userAgent:** Returns a string representing the user agent header sent by the browser. This string often contains information about the browser's name and version.
     ```
     const userAgent = navigator.userAgent;
     console.log(userAgent);
     ```
  2. **navigator.appName:** Returns the name of the browser.
     ```
     const browserName = navigator.appName;
     console.log(browserName);
     ```
  3. **navigator.appVersion:** Returns the version of the browser.
     ```
     const browserVersion = navigator.appVersion;
     console.log(browserVersion);
     ```
  4. **navigator.platform:** Returns the name of the client's operating system.
     ```
     const platform = navigator.platform;
     console.log(platform);
     ```
  5. **navigator.language**: Returns the preferred language of the user's browser.
     ```
     const preferredLanguage = navigator.language;
     console.log(preferredLanguage);
     ```
  6. **navigator.cookieEnabled`:** Returns a Boolean indicating whether cookies are enabled in the user's browser.
     ```
     const cookiesEnabled = navigator.cookieEnabled;
     console.log(cookiesEnabled);
     ```
  7. **navigator.onLine:** Returns a Boolean indicating whether the browser is currently online (connected to the internet).
     ```
     const onlineStatus = navigator.onLine;
     console.log(onlineStatus);
     ```

## ❖ Methods:

1. navigator.geolocation Allows access to the user's geographic position if the user grants permission. This can be used for location-based services.

```
if ('geolocation' in navigator) {
  navigator.geolocation.getCurrentPosition(function(position) {
   console.log('Latitude: ' + position.coords.latitude);
   console.log('Longitude: ' + position.coords.longitude);
  });
 } else {
  console.log('Geolocation is not supported in this browser.');
 }
```

2. **navigator.vibrate():** Allows the device to vibrate if supported.

```
if ('vibrate' in navigator) {
  navigator.vibrate(200); // Vibrate for 200 milliseconds
 } else {
  console.log('Vibration is not supported in this browser.');}
```

3. **navigator.clipboard:** Provides access to the clipboard API, allowing you to read and write to the clipboard if the user grants permission.

```
if ('clipboard' in navigator) {
  navigator.clipboard.writeText('Text to copy to clipboard').then(function() {
    console.log('Text copied to clipboard');
  }).catch(function(err) {
    console.error('Failed to copy text: ', err);
  });
} else {
  console.log('Clipboard API is not supported in this browser.');
}
```

**Example:**

```
<!DOCTYPE html>
<html><head>
 <title>JavaScript Navigator Object Example</title>
</head><body>
 <h1>Navigator Object Example</h1>
 <p><strong>User Agent:</strong> <span id="userAgentInfo"></span></p>
 <p><strong>Browser Name:</strong> <span id="browserNameInfo"></span></p>
 <p><strong>Browser Version:</strong> <span id="browserVersionInfo"></span></p>
 <p><strong>Platform:</strong> <span id="platformInfo"></span></p>
 <p><strong>Preferred Language:</strong> <span id="languageInfo"></span></p>
 <p><strong>Cookies Enabled:</strong> <span id="cookiesEnabledInfo"></span></p>
 <p><strong>Online Status:</strong> <span id="onlineStatusInfo"></span></p>
 <script>
  // Accessing properties of the Navigator object
  document.getElementById('userAgentInfo').textContent = navigator.userAgent;
  document.getElementById('browserNameInfo').textContent = navigator.appName;
  document.getElementById('browserVersionInfo').textContent = navigator.appVersion;
  document.getElementById('platformInfo').textContent = navigator.platform;
  document.getElementById('languageInfo').textContent = navigator.language;
  document.getElementById('cookiesEnabledInfo').textContent = navigator.cookieEnabled ?
'Enabled' : 'Disabled';
  document.getElementById('onlineStatusInfo').textContent = navigator.onLine ? 'Online' :
'Offline';
 </script></body></html>
```

**output:**

Navigator Object Example
User Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/116.0.0.0 Safari/537.36
Browser Name: Netscape
Browser Version: 5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/116.0.0.0 Safari/537.36
Platform: Win32
Preferred Language: en-US
Cookies Enabled: Enabled
Online Status: Online

# JAVASCRIPT SCREEN OBJECT:

- The JavaScript Screen object is a built-in object that provides information about the user's screen or display. It contains properties that allow you to retrieve details such as screen dimensions, color depth, and whether the user's screen is a mobile device or a desktop monitor.

## ❖ Properties:

1. **screen.width:** Returns the width of the screen in pixels.
   const screenWidth = screen.width;
2. **screen.height:** Returns the height of the screen in pixels.
   const screenHeight = screen.height;
3. **screen.availWidth:** Returns the available width of the screen in pixels, excluding any system taskbars or docks.
   const availableScreenWidth = screen.availWidth;
4. **screen.availHeight:** Returns the available height of the screen in pixels, excluding any system taskbars or docks.
   const availableScreenHeight = screen.availHeight;
5. **screen.colorDepth:** Returns the color depth of the screen in bits per pixel. This indicates the number of colors a screen can display.
   const colorDepth = screen.colorDepth;
6. **screen.pixelDepth:** Similar to `colorDepth`, this property also returns the color depth of the screen.
   const pixelDepth = screen.pixelDepth;
7. **screen.orientation:** Returns an object that represents the orientation of the screen. It includes properties like `type` (e.g., "landscape-primary" or "portrait-secondary") and `angle` (the rotation angle in degrees).
   const orientation = screen.orientation;
8. **screen.devicePixelRatio:** Returns the ratio of the physical pixels to CSS pixels on the screen. It's often used for responsive design and detecting high-density displays (e.g., Retina displays).
   const pixelRatio = screen.devicePixelRatio;
9. **screen.mobile:** A non-standard property that indicates whether the user's device is a mobile device. It's not supported in all browsers and is generally less reliable than other methods for detecting mobile devices.
   const isMobile = screen.mobile;

**Example:**

```
<!DOCTYPE html>
<html>
<head>
  <title>JavaScript Screen Object Example</title>
</head>
<body>
  <h1>Screen Object Example</h1>
  <p><strong>Screen Width:</strong> <span id="screenWidth"></span> pixels</p>
  <p><strong>Screen Height:</strong> <span id="screenHeight"></span> pixels</p>
  <p><strong>Available Width:</strong> <span id="availWidth"></span> pixels</p>
  <p><strong>Available Height:</strong> <span id="availHeight"></span> pixels</p>
  <p><strong>Color Depth:</strong> <span id="colorDepth"></span> bits per pixel</p>
```

```
<p><strong>Pixel Depth:</strong> <span id="pixelDepth"></span> bits per pixel</p>
<p><strong>Device Pixel Ratio:</strong> <span id="pixelRatio"></span></p>
<p><strong>Orientation:</strong> <span id="orientationType"></span></p>
<p><strong>Orientation Angle:</strong> <span id="orientationAngle"></span> degrees</p>
<script>
  // Accessing properties of the Screen object
  document.getElementById('screenWidth').textContent = screen.width;
  document.getElementById('screenHeight').textContent = screen.height;
  document.getElementById('availWidth').textContent = screen.availWidth;
  document.getElementById('availHeight').textContent = screen.availHeight;
  document.getElementById('colorDepth').textContent = screen.colorDepth;
  document.getElementById('pixelDepth').textContent = screen.pixelDepth;
  document.getElementById('pixelRatio').textContent = screen.devicePixelRatio;
  // Accessing screen orientation properties
  const orientation = screen.orientation;
  document.getElementById('orientationType').textContent = orientation.type;
  document.getElementById('orientationAngle').textContent = orientation.angle;
</script>
</body>
</html>
```

**output:**

Screen Object Example
Screen Width: 1280 pixels
Screen Height: 800 pixels
Available Width: 1280 pixels
Available Height: 752 pixels
Color Depth: 24 bits per pixel
Pixel Depth: 24 bits per pixel
Device Pixel Ratio:
Orientation: landscape-primary
Orientation Angle: 0 degrees

## ❖ Summary of BOM:

- The Browser Object Model (BOM)
  1. **Window Object:** The `window` object is the top-level object in the BOM hierarchy and represents the browser window. It provides access to various properties and methods for controlling and manipulating the browser window.
  2. **Document Object:** The `document` object represents the web page loaded in the browser window. It allows access to the document's structure and content, including elements and their properties.
  3. **Location Object:** The `location` object represents the URL of the current page and provides methods to navigate to different URLs. It allows you to read and manipulate the browser's address bar.
  4. **Navigator Object:** The `navigator` object provides information about the user's browser and operating system. It includes properties like `userAgent`, `appName`, `appVersion`, and more.

5. **Screen Object:** The `screen` object provides information about the user's screen or display, including properties like `width`, `height`, `colorDepth`, and `pixelDepth`.
6. **History Object:** The `history` object allows you to navigate through the browser's history stack, enabling actions like going back and forward between visited pages.
7. **Popup Windows:** JavaScript can open new browser windows or pop-up windows using methods like `window.open()`. These windows are represented as `window` objects themselves.
8. **Timers:** JavaScript can use timers like `setTimeout()` and `setInterval()` to schedule code execution at specific intervals or after a delay.
9. **Cookies:** The BOM includes methods for working with cookies, such as `document.cookie`, which allows you to read and write cookies in the browser
10. **Alerts, Prompts, and Confirmations:** JavaScript can display alerts, prompts, and confirmation dialogs using the `window.alert()`, `window.prompt()`, and `window.confirm()` methods.
11. **Event Handling:** The BOM enables event handling for user interactions and browser events. You can attach event listeners to various BOM objects to respond to events like clicks, mouse movements, and keyboard input.
12. **Local Storage and Session Storage:** BOM includes the `localStorage` and `sessionStorage` objects, which allow you to store data on the client-side persistently (localStorage) or for the duration of a session (sessionStorage).

# COOKIES

- Cookies in JavaScript are small pieces of data that websites can store on a user's device.
- They are often used for various purposes, such as tracking user sessions, storing user preferences, and maintaining stateful information between HTTP requests.
- A cookie is an amount of information that persists between a server-side and a client-side.

## ❖ How Cookies Works?

- step-by-step explanation of how cookies work:
  - **Server-Side Creation:** When a user interacts with a website, the web server can create a cookie by including a Set-Cookie HTTP response header in its response to the user's browser. This header contains information about the cookie, including its name, value, and various optional attributes.
- For example, a server can respond with a Set-Cookie header like this:

  Set-Cookie: username=Sangam; expires=Thu, 01 Jan 2025 00:00:00 UTC; path=/; domain=mywebsite.com; secure
  - **Storage in the Browser:** Once received, the browser stores this cookie data locally on the user's device. The data is typically stored in a text file or a similar data structure.
  - **Automatic Sending:** From this point on, whenever the user makes an HTTP request to the same domain (e.g., by clicking on links or loading pages), the browser automatically includes all relevant cookies for that domain in the Cookie HTTP request header.
- For example, if the user visits another page on the same website, the browser sends an HTTP request like this:

GET /somepage HTTP/1.1
  - **Host:** mywebsite.com
  - **Cookie:** username=Sangam; othercookie=value
- **note:** The Cookie header contains all the cookies associated with that domain.
  - **Server-Side Usage:** On the server, when it receives an HTTP request, it can access the Cookie header to retrieve the cookies sent by the user's browser. The server can then read and interpret the cookie data to determine, for example, the user's identity, preferences, or session information.
  - **Updating and Deleting Cookies:** The server can also update or delete cookies by sending new Set-Cookie headers with the HTTP response. To delete a cookie, the server typically sets its expiration date to a date in the past.
  - **Client-Side Access:** Cookies can also be accessed and manipulated on the client-side using JavaScript. This allows web developers to read and modify cookies as needed for client-side functionality, such as remembering user preferences or managing client-side sessions.

**Example:**

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
<input type="button" value="setCookie" onclick="setCookie()">
<input type="button" value="getCookie" onclick="getCookie()">
  <script>
  function setCookie()
```

```
    {
      document.cookie="username=Sangam Kumar";
    }
    function getCookie()
    {
      if(document.cookie.length!=0)
      {
      alert(document.cookie);
      }
      else
      {
      alert("Cookie not available");
      } }
    </script>
  </body>
  </html>
```

**Exampe-2:**

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
    <select id="color" onchange="display()">
        <option value="Select Color">Select Color</option>
        <option value="yellow">Yellow</option>
        <option value="green">Green</option>
        <option value="red">Red</option>
      </select>
      <script type="text/javascript">
        function display()
        {
          var value = document.getElementById("color").value;
          if (value != "Select Color")
          {
            document.bgColor = value;
            document.cookie = "color=" + value;
          } }
        window.onload = function ()
        {
          if (document.cookie.length != 0)
          {
            var array = document.cookie.split("=");
            document.getElementById("color").value = array[1];
            document.bgColor = array[1];
          } }
      </script>
</body>
</html>
```

## ❖ cookies attributes:

- Cookies in JavaScript can have various attributes that control their behavior.

➢ These attributes are specified when creating a cookie using the `document.cookie` property.

**1. Name and Value:**

- name=value: The name and value pair is the basic content of a cookie. It represents the data associated with the cookie.

**2. Expires:**

- expires=date: Specifies the expiration date and time for the cookie. After this date and time, the cookie will be deleted. The date should be in the format "Day, DD Mon YYYY HH:MM:SS GMT."
- document.cookie = "username=John; expires=Thu, 01 Jan 2025 00:00:00 GMT";

**3. Max-Age:**

- max-age=seconds: Specifies the maximum age of the cookie in seconds. After the specified number of seconds, the cookie will expire.

document.cookie = "sessionToken=12345; max-age=3600"; // Expires in 1 hour

**4. Path:**

- path=path: Defines the path for which the cookie is valid. The cookie will only be sent to the server for URLs that match this path.
- document.cookie = "theme=dark; path=/myapp";

**5. Domain:**

- domain=domain: Specifies the domain for which the cookie is valid. By default, cookies are associated with the domain of the current page.
- document.cookie = "auth=123; domain=mywebsite.com";

**6. Secure:**

- secure: When present, the cookie will only be sent over secure (HTTPS) connections. It helps protect sensitive information.

document.cookie = "secureCookie=secret; secure";

**7. HttpOnly:**

- HttpOnly: When present, the cookie cannot be accessed by JavaScript. This attribute enhances security by preventing client-side scripts from reading the cookie.
- document.cookie = "sessionId=abc123; HttpOnly";

**8. SameSite:**

- SameSite: Specifies how cookies should be sent in cross-origin requests. This attribute helps prevent certain types of CSRF attacks. Possible values are `"Strict"`, `"Lax"`, or `"None"`.
- document.cookie = "csrfToken=xyz; SameSite=Strict";

➢ These attributes can be combined when setting a cookie to control its behavior. For example:

- document.cookie = "myCookie=value; expires=Thu, 01 Jan 2025 00:00:00 GMT; path=/; domain=mywebsite.com; secure; HttpOnly; SameSite=Strict";

## ❖ Deleting a Cookie:

- You can delete a cookie in JavaScript by setting its expiration date to a past date.
- When a cookie's expiration date is in the past, the browser will automatically remove it.
- Different ways to delete a Cookie
- These are the following ways to delete a cookie:
- A cookie can be deleted by using expire attribute.
- A cookie can also be deleted by using max-age attribute.
- We can delete a cookie explicitly, by using a web browser.

**Example:**

```
// Create a cookie
document.cookie = "username=raj";
// Delete the cookie by setting its expiration date to the past
document.cookie = "username=; expires=Thu, 01 Jan 2010 00:00:00 GMT";
Example:
<!DOCTYPE html>
<html>
<head>
</head>
<body>
<input type="button" value="Set Cookie" onclick="setCookie()">
<input type="button" value="Get Cookie" onclick="getCookie()">
<script>
function setCookie()
{
   document.cookie="name=Martin Roy; expires=Sun, 20 Aug 2000 12:00:00 UTC";

}
function getCookie()
{
   if(document.cookie.length!=0)
   {
   alert(document.cookie);
   }
   else
   {
      alert("Cookie not avaliable");
   }
}
</script>
</body>
</html>
```

# JAVASCRIPT DEBUGGING:

- JavaScript debugging is the process of identifying and fixing errors or issues in your JavaScript code. Debugging tools and techniques help you examine the code's behavior, trace the flow of execution, and pinpoint the source of problems.
- Here are common techniques and tools for debugging JavaScript:

**1. Console Logging:**

- Use `console.log()`, `console.error()`, and `console.warn()` to print values, variables, and messages to the browser's console. This is a quick and effective way to inspect the state of your code.

- console.log("Debugging message:", someVariable);

**2. Breakpoints:**

- Set breakpoints in your code using the browser's developer tools. Breakpoints pause execution at a specific line, allowing you to inspect variables and step through code.

**3. Step Through Code:**

- Use the debugging tools to step through code one line at a time. You can step into functions, step over function calls, and step out of functions to understand the flow of execution.

**4. Inspect Variables:**

- Examine the values of variables and objects in the scope. Most debugging tools provide a "Variables" or "Scope" panel where you can inspect and modify variables.

**5. Conditional Breakpoints:**

- Set breakpoints that only trigger when a specific condition is met. This can help you debug issues that occur under certain circumstances.

**6. Call Stack:**

- The call stack shows the sequence of function calls that led to the current point in your code. Understanding the call stack can help you identify where an error occurred.

**7. Console Assertions:**

- Use `console.assert()` to check if a condition is true and log an error message if it's not. This is helpful for verifying assumptions in your code.

- console.assert(x > 0, "x should be greater than 0");

**8. Try-Catch Blocks:**

- Wrap code in try-catch blocks to catch and handle exceptions gracefully. This prevents errors from crashing your entire application.

```
try {
  // Code that may throw an error
} catch (error) {
  // Handle the error
}
```

**9. Debugger Statement:**

- Insert the `debugger` statement in your code to force a breakpoint at a specific location. This is useful for triggering debugging when needed.

```
function someFunction() {
  // ...
  debugger; // Pause execution here
  // ...
}
```

**10. Error Messages and Stack Traces:**

- Pay attention to error messages and stack traces in the console. They provide valuable information about what went wrong and where.

**11. Browser Developer Tools:**

- Each web browser provides developer tools that include debugging features. Commonly used browsers like Chrome, Firefox, and Edge have robust developer toolsets for JavaScript debugging.

**12. External Debugging Tools:**

- Consider using external JavaScript debugging tools like Visual Studio Code (with browser extensions), Firefox Developer Edition, or dedicated IDEs for JavaScript development.

**Example:**

- examples of common JavaScript debugging scenarios along with techniques to address them using various debugging methods:

**1. Syntax Error:**

```
// Example with a syntax error
const greeting = "Hello";
console.log(greeting; // Missing closing parenthesis
// Debugging: You'll see a syntax error in the browser's console.
```

- In this case, the browser's console will display a syntax error, and you can easily spot the issue in the code.

**2. Variable Inspection:**

```
// Example with variable inspection
let x = 5;
let y = 10;
let z = x + y;
// Debugging: Place a breakpoint at the line with "let z = x + y;" and inspect the values of x, y, and z in the debugger's Variables panel.
```

- Set a breakpoint at the line of interest and inspect the variable values during runtime using the debugger.

## 3. Console Logging:

```
// Example with console logging
function divide(a, b) {
  console.log(`Dividing ${a} by ${b}`);
  return a / b;
}
divide(10, 0); // Division by zero
// Debugging: Use console.log to print debug information. In this case, you'll see the log message before the error.
```

- Use `console.log()` to log messages and variable values to the console for debugging purposes.

**4. Conditional Breakpoints:**

```
// Example with a conditional breakpoint
function findElement(arr, target) {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) {
      console.log(`Found ${target} at index ${i}`);
```

```
      return i;
    }
   }
   return -1;
  }
  const numbers = [1, 3, 5, 7, 9];
  findElement(numbers, 5); // Debugging: Set a breakpoint with a condition (e.g., i === 2) to stop execution when i is 2.
```

- Set a conditional breakpoint to pause code execution when specific conditions are met, allowing you to inspect the program state at that point.

**5. Try-Catch for Error Handling:**

```
  // Example with try-catch for error handling
  try {
   // Code that may throw an error
   const result = someFunction();
   console.log(`Result: ${result}`);
  } catch (error) {
   console.error(`An error occurred: ${error.message}`);
  }

  // Debugging: Wrap code that might throw an error in a try-catch block to handle and log errors gracefully.

  function someFunction() {
   return undefinedVariable; // This will throw a ReferenceError
  }
```

- Use try-catch blocks to catch and handle errors, allowing your code to continue executing even if an error occurs.

**6. Using the `debugger` Statement:**

```
  // Example using the debugger statement
  function complexCalculation(a, b) {
   let result = a * b;
   debugger; // Set a breakpoint here
   result = result + 10;
   return result;
  }
  const result = complexCalculation(5, 3);
  // Debugging: When the code reaches the "debugger" statement, execution will pause, and you can inspect variable values and the call stack.
```

- Insert the `debugger` statement in your code to force a breakpoint at a specific location for detailed debugging.

# JAVASCRIPT PROMISES:

- JavaScript Promises are a way to handle asynchronous operations in a more structured and manageable manner.
- Promises in real-life express a trust between two or more persons and an assurance that a particular thing will surely happen.
- In javascript, a Promise is an object which ensures to produce a single value in the future (when required).
- They provide a way to represent a value that may not be available yet but will be resolved at some point in the future, either successfully or with an error.
- Promises have become a fundamental part of JavaScript for dealing with asynchronous tasks like fetching data from a server, reading files, or handling user interactions.

❖ **A Promise can have three states:**

1. **Pending:** The initial state, indicating that the Promise is still waiting for the result.
2. **Fulfilled (Resolved):** The state when the asynchronous operation is successfully completed, and the Promise holds a resolved value.
3. **Rejected:** The state when an error occurs during the asynchronous operation, and the Promise holds a reason for the rejection.

## ❖ how to use Promises in JavaScript:

➢ Creating a Promise

- You can create a Promise using the Promise constructor, which takes a single argument, a function called the executor. The executor function has two parameters: resolve and reject. You call these functions to indicate whether the Promise should be fulfilled or rejected.

**Example:**

```
const myPromise = new Promise((resolve, reject) => {
 // Asynchronous operation
 setTimeout(() => {
  const randomNumber = Math.random();
  if (randomNumber > 0.5) {
   resolve(randomNumber); // Resolve with a value
  } else {
   reject("Error: Number too small"); // Reject with an error message
  }
 }, 1000); // Simulating an asynchronous operation
});
```

Using .then() and .catch()

- Once you have a Promise, you can use the .then() method to specify what to do when the Promise is fulfilled, and you can use the .catch() method to specify what to do when the Promise is rejected.

## Example:

```
myPromise
 .then((result) => {
  console.log("Promise fulfilled with result:", result);
 })
 .catch((error) => {
  console.error("Promise rejected with error:", error);
```

```
 });
Example with Output
// Creating a Promise
const myPromise = new Promise((resolve, reject) => {
 // Simulating an asynchronous operation
 setTimeout(() => {
  const randomNumber = Math.random();
  if (randomNumber > 0.5) {
   resolve(randomNumber); // Resolve with a value
  } else {
   reject("Error: Number too small"); // Reject with an error message
  }
 }, 1000);
});

// Using .then() and .catch() to handle the Promise
myPromise
 .then((result) => {
  console.log("Promise fulfilled with result:", result);
 })
 .catch((error) => {
  console.error("Promise rejected with error:", error);
 });
```

**Output:**

Promise fulfilled with result: [random number]
If the random number is 0.5 or smaller, you'll see:
Promise rejected with error: Error: Number too small

**Note:** In this example, the Promise simulates an asynchronous operation with a random outcome.
Asynchronous in JavaScript:

- Asynchronous is a programming paradigm that allows you to execute code independently of the main program flow.
- It enables non-blocking operations, meaning that instead of waiting for an operation to complete, the program can continue executing other tasks. Asynchronous JavaScript is essential for handling tasks like network requests, file reading, timers, and user interactions without freezing the user interface.

➢ Here are key concepts and techniques for asynchronous programming in JavaScript

**1. Callbacks:**

- Callbacks are functions passed as arguments to other functions. They are executed when an asynchronous operation is complete.

**Example 1:** Using Callbacks

```
function fetchData(callback) {
 setTimeout(() => {
  callback("Data received");
 }, 1000);
}
fetchData((result) => {
```

```
  console.log(result); // Output: Data received
});
```

- In this example, the `fetchData` function simulates an asynchronous operation using `setTimeout` and calls the provided callback when it's done.

**2. Promises:**

- Promises are a more structured way to handle asynchronous operations and provide better error handling.

**Example 2:** Using Promises

```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const success = true;
      if (success) {
        resolve("Data received");
      } else {
        reject("Error: Data not received");
      }
    }, 1000);
  });
}

fetchData()
  .then((result) => {
    console.log(result); // Output: Data received
  })
  .catch((error) => {
    console.error(error); // Output: Error: Data not received
  });
```

In this example, the `fetchData` function returns a Promise that resolves when the data is received and rejects if there's an error.

## 3. Async/Await:

- Async/await is a more readable way to work with Promises and make asynchronous code look synchronous.

**Example 3:** Using Async/Await

```
async function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const success = true;
      if (success) {
        resolve("Data received");
      } else {
        reject("Error: Data not received");
      }
    }, 1000);
  });
}
```

```
async function main() {
  try {
    const result = await fetchData();
    console.log(result); // Output: Data received
  } catch (error) {
    console.error(error); // Output: Error: Data not received
  }
}
main();
```

- In this example, the `main` function uses the `await` keyword to wait for the Promise to resolve or reject, making the code appear synchronous.

## 4. Callbacks vs. Promises vs. Async/Await:

- Callbacks are less structured and can lead to callback hell or the pyramid of doom when dealing with multiple async operations. Promises and async/await provide cleaner and more maintainable code.

## 5. Event Loop:

- The event loop is a central part of how asynchronous JavaScript works. It manages the execution of asynchronous tasks and callbacks.

**6. Timers:**

- Timers like `setTimeout` and `setInterval` are commonly used for scheduling code to run asynchronously.

**Example 4:** Using setTimeout

```
console.log("Start");
setTimeout(() => {
  console.log("Inside setTimeout");
}, 1000);
console.log("End");
// Output:
// Start
// End
// Inside setTimeout (after 1 second)
```

**Note:** Asynchronous JavaScript is crucial for building responsive and efficient web applications, especially when dealing with I/O operations and interactions with external resources.

## Complete Example:

### Index.js:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Promise Example</title>
   <link rel="stylesheet" href="Raj.css">
</head>
<body>
```

```
    <div class="container">
      <h1>Random User Data</h1>
      <button id="fetchButton">Fetch Data</button>
      <div id="userData">
        <!-- User data will be displayed here -->
      </div>
    </div>
    <script src="Raj.js"></script>
  </body>
  </html>
```

- **Raj.js**

```
// Function to fetch random user data using a Promise
function fetchUserData() {
  return new Promise((resolve, reject) => {
    fetch('https://jsonplaceholder.typicode.com/users/1')
      .then((response) => {
        if (!response.ok) {
          throw new Error('Network response was not ok');
        }
        return response.json();
      })
      .then((data) => {
        resolve(data);
      })
      .catch((error) => {
        reject(error);
      });
  });
}
// Function to display user data on the web page
function displayUserData(userData) {
  const userDataDiv = document.getElementById('userData');
  userDataDiv.innerHTML = `
    <h2>User Data</h2>
    <p><strong>Name:</strong> ${userData.name}</p>
    <p><strong>Email:</strong> ${userData.email}</p>
    <p><strong>Phone:</strong> ${userData.phone}</p>
    <p><strong>Website:</strong> ${userData.website}</p>
  `;
}
// Add an event listener to the Fetch Data button
const fetchButton = document.getElementById('fetchButton');
fetchButton.addEventListener('click', () => {
  fetchUserData()
    .then((userData) => {
      displayUserData(userData);
    })
```

```
    .catch((error) => {
      const userDataDiv = document.getElementById('userData');
      userDataDiv.innerHTML = `<p>Error: ${error.message}</p>`;
    });
});
```

- **Raj.css**

```
body {
  font-family: Arial, sans-serif;
  background-color: #f2f2f2;
  margin: 0;
  padding: 0;
}
.container {
  background-color: #fff;
  max-width: 400px;
  margin: 0 auto;
  padding: 20px;
  border-radius: 5px;
  box-shadow: 0 0 5px rgba(0, 0, 0, 0.2);
}
h1 {
  text-align: center;
}
button {
  background-color: #007bff;
  color: #fff;
  padding: 10px 20px;
  border: none;
  border-radius: 3px;
  cursor: pointer;
  display: block;
  margin: 0 auto;
}
button:hover {
  background-color: #0056b3;
}
#userData {
  margin-top: 20px;
  padding: 10px;
  border: 1px solid #ccc;
  border-radius: 3px;
  background-color: #f9f9f9;
}
```

# DOCUMENT OBJECT MODEL(DOM)

## ❖ What is the DOM?

- The Document Object Model (DOM) is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content dynamically.

## ❖ The Virtual DOM:

- React introduces the concept of a "virtual DOM." Instead of directly manipulating the actual DOM, React creates and maintains a lightweight virtual representation of it in memory. This virtual DOM is a tree-like structure that mirrors the actual DOM's structure.

## ❖ Tree-like Structure:

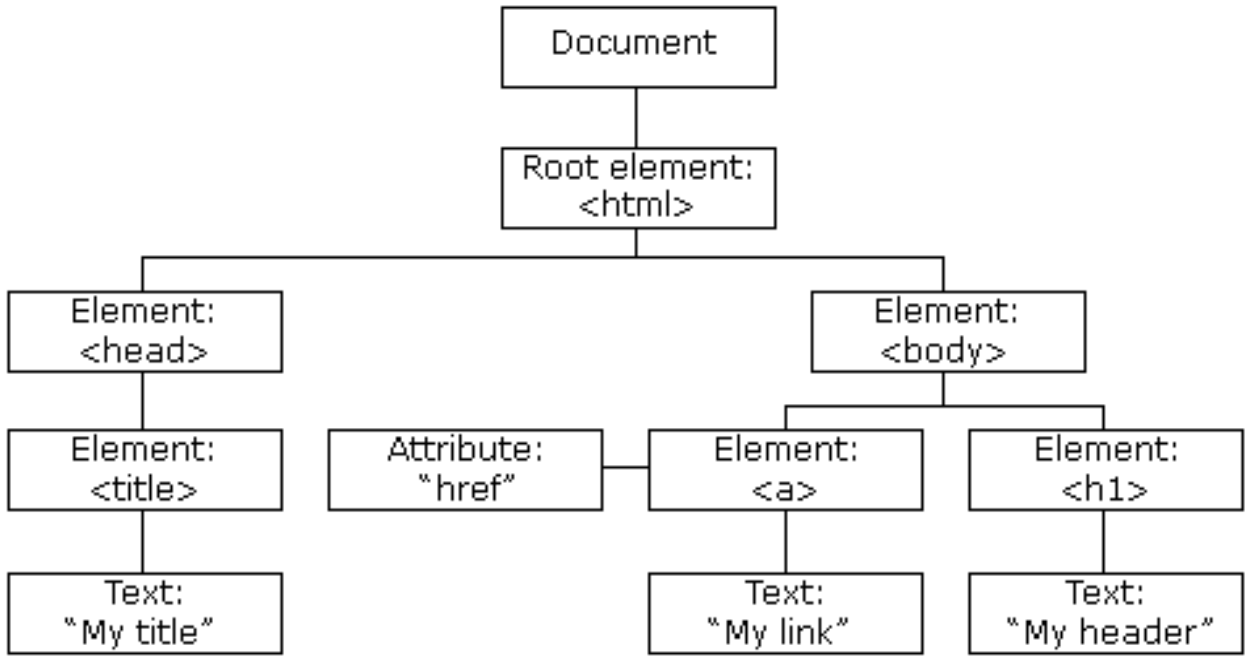
- The DOM represents an HTML or XML document as a hierarchical tree structure. Each element in the document, such as HTML tags, text, attributes, and comments, is represented as a node in this tree. These nodes are organized in a parent-child relationship, where the document itself is the root node.

## ❖ Node Types:

➢ There are several types of nodes in the DOM, including:

- **Element Nodes:** Represent HTML elements like , <p>, or <a>.
- **Text Nodes:** Contain text within an element.
- **Attribute Nodes:** Store attributes of elements.
- **Comment Nodes:** Contain comments within the HTML.
- **Document Node:** Represents the entire HTML document.

# HTML DOM Tree of Objects



## ❖ What is the HTML DOM?

- The HTML DOM is a standard object model and programming interface for HTML. It defines:
  - The HTML elements as objects
  - The properties of all HTML elements
  - The methods to access all HTML elements
  - The events for all HTML elements
- In other words: HTML DOM is a standard for how to get, change, add, or delete HTML elements.

**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>
</head>
<body>
  <h1>POWER OF DOM</h1>
  <p id="one"><b> With the object model, JavaScript gets all the power it needs to create dynamic HTML:</b></p>
  <ul>
<li>JavaScript can change all the HTML elements in the page</li>
<li>JavaScript can change all the HTML attributes in the page</li>
<li>JavaScript can change all the CSS styles in the page</li>
<li>JavaScript can remove existing HTML elements and attributes</li>
<li>JavaScript can add new HTML elements and attributes</li>
<li>Javascript can react to all existing HTML events in the page</li>
<li>JavaScript can create new HTML events in the page</li>
  </ul>
  <script>
    console.log(document)
    // With the HTML DOM, JavaScript can access and change all the elements of an HTML document
    console.log(document.getElementById("one"))
    console.log(document.getElementById("one").innerHTML)
    console.log(document.getElementById("one").innerText)
    document.getElementById("one").innerText="JS DOM features---"
  </script>
</body>
</html>
```

# HTML DOM METHODS

- HTML DOM methods are actions.
- HTML DOM properties are values

**Note:** The HTML DOM can be accessed with JavaScript (and with other programming languages).

- The programming interface is the properties and methods of each object.
  - A **property is a value** that you can get or set (like changing the content of an HTML element).
  - A **method is an action** you can do (like add or deleting an HTML element).

## ❖ Methods for Selecting Elements:

1. **document.getElementById(id):** Selects an element by its id attribute.
2. **document.getElementsByClassName(className):** Returns a live HTMLCollection of elements with the specified class name.
3. **document.getElementsByTagName(tagName):** Returns a live HTMLCollection of elements with the specified tag name.
4. **document.querySelector(selector):** Selects the first element that matches the CSS selector.
5. **document.querySelectorAll(selector):** Selects all elements that match the CSS selector.
6. **document.createElement(tagName**): Creates a new HTML element with the specified tag name.
7. **document.createTextNode(text):** Creates a new text node with the specified text content.

**Example-1:**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <h1 id="one">DOM METHODS FOR FINDING HTML ELEMENTS</h1>
  <P class="a">1.document.getElementById</P>
  <p class="a">2.document.getElementsByTagName</p>
  <p class="b">3.document.getElementsByClassName</p>
  <p>4.document.querySelector</p>
  <p>5.document.querySelectorAll</p>
  <h6 class="a">hello</h6>
  <script>
    // console.log(document.getElementById("one"))
    // console.log(document.getElementsByTagName("p")[1])
    // getElementsByTagName returns a collection of matching Elements. To access individual element, you need to use indexing process
    // console.log(document.getElementsByClassName("a")[0])
// getElementsByClassName returns a collection of matching Elements by class name. To access individual element, you need to use indexing process
//  console.log(document.querySelector("p.a"))
//  console.log(document.querySelector("h1#one"))
```

```
//  console.log(document.querySelector(".a"))
 // querySelector returns a single element matching by given combination.
 console.log(document.querySelectorAll(".a"))
 console.log(document.querySelectorAll(".a")[0])
 console.log(document.querySelectorAll("p.a"))
 console.log(document.querySelectorAll("p.a")[1])
 console.log(document.querySelectorAll("#one")[0])
 //querySelectorAll  returns a collection of matching Elements by given combnation. To access
individual element, you need to use indexing process
  </script>
</body>
</html>
```

**Example-2:**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>DOM Selection Example</title>
</head>
<body>
  <div id="myElement">This is a div element with an ID.</div>
  <p class="myClass">Paragraph 1</p>
  <p class="myClass">Paragraph 2</p>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
  </ul>
  <script>
    // getElementById
    const elementById = document.getElementById('myElement');
    console.log('getElementById:');
    console.log(elementById.textContent);
    // getElementsByClassName
    const elementsByClass = document.getElementsByClassName('myClass');
    console.log('\ngetElementsByClassName:');
    for (const element of elementsByClass) {
      console.log(element.textContent);
    }
    // getElementsByTagName
    const elementsByTagName = document.getElementsByTagName('li');
    console.log('\ngetElementsByTagName:');
    for (const element of elementsByTagName) {
      console.log(element.textContent);
    }
    // querySelector
    const querySelector = document.querySelector('.myClass');
```

```
        console.log('\nquerySelector:');
        console.log(querySelector.textContent);
        // querySelectorAll
        const querySelectorAll = document.querySelectorAll('ul li');
        console.log('\nquerySelectorAll:');
        for (const element of querySelectorAll) {
          console.log(element.textContent);
        }
      </script>
    </body>
    </html>
```

# Changing Element Content:

1. **textContent:** Set or get the plain text content of an element.
   - innerHTML: Set or get the HTML content of an element.
2. **Modifying Element Attributes:**
   - setAttribute(name, value): Change or add an attribute to an element.
3. **Changing Element Styles:**
   - style.property = value: Modify an element's CSS properties.
4. **Creating and Appending New Elements:**
   - document.createElement(tagName): Create a new HTML element.
   - element.appendChild(newChild): Append a child node to an element.
5. **Removing Elements:**

- element.remove(): Remove an element from the DOM.

6. **Replacing Elements:**

- element.replaceWith(newElement): Replace an element with another.

**Example-1:**

- **dom_chaningelements.html:**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <h1 id="one">Changing HTML Elements</h1>
  <ol>
    <li>element.innerHTML =  new html content</li>
    <li>element.attribute = new value</li>
    <li>element.style.property = new style</li>
    <li>element.setAttribute(attribute, value)</li>
  </ol>
  <p>First three are properties and the fourth one is method</p>
  <p id="two">dummy</p>
<script>
```

```
    document.getElementById("one").innerHTML="<i>These are the few ways for changing HTML elements</i>"
    // Placing the i tag is possible here because of innerHTML, it accepts content and tag behavior also
    document.getElementById("two").innerText="<b>Updated in JS</b>"
    //innerText accepts text content only, adding b tag will not apply its behavior here
</script>
</body>
</html>
```

## Example-2:

- **dom_elementatribute.html:**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1>element.attribute = new value</h1>
    <a id="one" href="https://www.twksaa.org "> Original Link</a>
    <button onclick="myFun()">Click here</button>
    <script>
        function myFun(){
        let element=document.getElementById("one")
        element.href="https://www.t3skills.org/"
        element.innerText="Modified Link"
        }
    </script>
</body>
</html>
```

## Example-3:

- **dom_stylepropeties.html:**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1>element.style.property = new style</h1>
    <div id="one" style="width:100px;height:200px;background-color: yellow;"> </div>
    <button onclick="changeWidth()">Change Width</button>
    <button onclick="changeColor()">Change Color</button>
    <script>
        let element=document.getElementById("one")
```

```
        function changeWidth(){
          element.style.width="300px"
        }
        function changeColor(){
          element.style.backgroundColor="orange"
        }
      </script>
    </body>
    </html>
```

**Example-4:**

- Dom_setatributes.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
<style>
  .a{
    border: 5px solid red;
  }
  #two{
    background-color: yellow;
    color: blue;
    font-size: 20px;
  }
</style>
</head>
<body>
  <h1>element.setAttribute(attribute, value)</h1>
  <p id="one">Hello World</p>
  <button onclick="myFun()">Click-1</button>
  <button onclick="myFun1()">Click-2</button>
  <script>
    function myFun(){
    let element=document.getElementById("one")
   element.setAttribute("class","a")
    }
    function myFun1(){
    let element=document.getElementById("one")
   element.setAttribute("id","two")
    }
  </script>
</body>
</html>
```

**Example-5:**

- **dom_write.html:**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<style>
</style>
<body>
  <h1 id="one"> </h1>
  <h4 id="two">
    <button onclick="myFun()">Click to display</button>
  </h4>
  <script>
    let a=prompt("Enter the number ")
    document.getElementById("one").innerText="Multiplication table of "+a;
    let n=a;
    function myFun(){
    for(let i=1;i<10;i++){
      document.write(n," x ",i," = ",n*i,"<br>")
    }
    }
  </script>
</body>
</html>
```

**Example-6:**

- **dom_createElement.html:**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<style>
  #one{
    height:100px;
    width:100px;
    border:5px solid red;
  }
</style>
<body>
  <h1>document.createElement(element)</h1>
  <button onclick="myFun()">Click to add</button>
```

```
    <script>
      function myFun(){
        let a=document.createElement("div")
      a.id="one"
      document.body.appendChild(a)
      }
    </script>
  </body>
  </html>
```

**Example-7:**

- dom_remove.html:

```
    <!DOCTYPE html>
    <html lang="en">
    <head>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
      <title>Document</title>
    </head>
    <style>
      p{
        height:100px;
        width:100px;
        border: 5px solid red;
      }
    </style>
    <body>
      <h1>document.removeChild(element)</h1>
    <p >
    </p>
    <button onclick="myFun()">Remove div</button>
    <script>
      let a=document.getElementsByTagName("p")[0]
      function myFun(){
        if(a){
          a.parentNode.removeChild(a)
        }
      }
    </script>
    </body>
    </html>
```

## Example-8:

- **dom-replacechild.html:**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<style>
  #one {
    height: 100px;
    width: auto;
    border: 5px solid red;
  }
  #two {
    height: 100px;
    width: auto;
    border: 5px solid blue;
  }
</style>
<body>
  <h1>document.replaceChild(new, old)</h1>
  <p id="one">This is old para </p>
  <button onclick="myFun()">Click to replace</button>
  <script>
    function myFun() {
      let a = document.getElementById("one")
      let b = document.createElement("p")
      b.id = "two"
      let txt = document.createTextNode("This is new para")
      b.appendChild(txt)
      a.parentNode.replaceChild(b, a)
    }
  </script>
</body>
</html>
```

## Example-9:

- **dom.addevent.html:**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
```

```
    <h1>Adding Events Handlers</h1>
    <p>Syntax:<br>
    document.getElementById(id).onclick = function(){code}
    </p>
    <button id="one">Click</button>
  <script>
    // document.getElementById("one").onclick = function(){
    //     alert("Button clicked successfully")
    // }
    let btn = document.getElementById("one")
    btn.onclick = function(){
      alert("Click event added")
    }
  </script>
  </body>
  </html>
```

**Example-10:**

- **dom_createtable.html**

```
    <!DOCTYPE html>
    <html lang="en">
    <head>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
      <title>Document</title>
    </head>
    <style>
      img{
        height:200px;
        width:200px;
      }
      table{
        width: 100%;
        text-align: center;
      }
    </style>
    <body>
      <table border="1">
        <thead>
          <th>Name</th>
          <th>Age</th>
          <th>Image</th>
        </thead>
        <tbody id="t-body">
        </tbody>
      </table>
      <script>
        let data = [
          { "name": "Raj", age: 20, image: "https://i.postimg.cc/qMdDpVSH/one.jpg" },
```

```
        { "name": "Aryan", age: 30, image: "https://i.postimg.cc/j28D3rJn/two.jpg" },
        { "name": "Kazi", age: 40, image: "https://i.postimg.cc/Pq2nwrGD/three.jpg" },
        { "name": "Peter", age: 31, image: "https://i.postimg.cc/L6t9sSst/four.jpg" }
      ]
      let t = document.getElementById("t-body")
      for (let i = 0; i < data.length; i++) {
        let row = t.insertRow(i)
        let cell1 = row.insertCell(0)
        let cell2 = row.insertCell(1)
        let cell3 = row.insertCell(2)
        let pic = document.createElement("img")
        pic.src = data[i].image
        cell1.innerText = data[i].name
        cell2.innerText = data[i].age
        cell3.appendChild(pic)
      }
    </script>
  </body>
  </html>
```

## Example-11:

- **multiplication table.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Tables</title>
</head>
<style>
#two{
  height:auto;
  width: auto;
  margin:0 auto;
  border: 2px solid gold;
  background-color: pink;
}
</style>
<body>
  <h1 id="one"> </h1>
    <button onclick="myFun()">Click to display</button>
  <script>
    let a=prompt("Enter the number ")
    document.getElementById("one").innerText="Multiplication table of "+a;
    let num=a;
    function myFun(){
      let n = document.createElement("div")
      n.id="two"
```

```
            document.body.appendChild(n)
            let op = ""
          for(let i=1;i<=10;i++){
            op=op+num+" x "+i+" = "+num*i+"<br>";
          }
          console.log(op)
          document.getElementById("two").innerHTML=op
          }
        </script>
      </body>
      </html>
```

## Example-12

- **Add event.html:**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <button id="btn">Click</button>
<script>
  let button = document.getElementById("btn")
  // Method-1
  // button.addEventListener("click",function(){
  //    alert("Click-1 done")
  // })
  //Method-2
  // button.addEventListener("click",()=>{
  //    alert("Click-2 done")
  // })
   //Method-3
  //  button.addEventListener("click",fun)
  //  function fun(){
  //    alert("Click 3 done")
  //  }
  //double click event
  button.addEventListener("dblclick",()=>{
    alert("Double click")
  })
</script>
</body>
</html>
```

**Example-13:**

- **Add_class .html**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>classList</title>
</head>
<style>
  .one{
    border:5px solid red;
  }
  .two{
    height: 100px;
    background-color: aqua;
  }
</style>
<body>
  <h1>Adding multiple classes to an element</h1>
  <p>className will be used to add single class to an element</p>
  <p>classList handles multiple class names at one time</p>
  <script>
    let p = document.createElement("p")
    p.innerText="Dummy para text"
    // p.className="one"
    p.classList.add("one")
    p.classList.add("two")
    document.body.appendChild(p)
  </script>
</body>
</html>
```

**Example-14:**

- **Todo_index.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>To-DO List</title>
</head>
<style>
  ol li{
    font-size: 30px;
  }
</style>
```

```
<body>
  <h1>ToDo List</h1>
  <div id="container">
    <input type="text" id="inp">
    <button id="add">Add</button>
    <ol id="todos"></ol>
  </div>
  <script>
    let inpF = document.getElementById("inp")
    let btn = document.getElementById("add")
    let todo = document.getElementById("todos")
    btn.addEventListener("click", () => {
      let res = document.createElement("li")
      res.innerText = inpF.value
      todo.appendChild(res)
      inpF.value = "" //clear the input field
      res.addEventListener("click",()=>{
        res.style.textDecoration = "line-through"
      })
      res.addEventListener("dblclick",()=>{
        todo.removeChild(res)
      })
    })
  </script>
</body>
</html>
```

**Example-15:**

- Todo_index2.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>To-DO List</title>
</head>
<style>
  ol li{
    font-size: 30px;
  }
  ol button{
   margin-left: 20px;
   margin-right: 20px;
  }
</style>
<body>
  <h1>ToDo List</h1>
  <div id="container">
    <input type="text" id="inp">
```

```
    <button id="add">Add</button>
    <ol id="todos"></ol>
  </div>
  <script>
    let inpF = document.getElementById("inp")
    let btn = document.getElementById("add")
    let todo = document.getElementById("todos")
    btn.addEventListener("click", () => {
      let res = document.createElement("li")
      let sp = document.createElement("span")
      sp.innerText = inpF.value
      res.appendChild(sp)
      let btn1 = document.createElement("button")
      btn1.innerText="Mark"
      res.appendChild(btn1)
      let btn2 = document.createElement("button")
      btn2.innerText="Remove"
      res.appendChild(btn2)
      todo.appendChild(res)
      inpF.value = "" //clear the input field
      btn1.addEventListener("click",()=>{
        res.style.textDecoration = "line-through"
      })
      btn2.addEventListener("click",()=>{
        todo.removeChild(res)
      })
    })
  </script>
</body>
</html>
```

## ❖ Adding and Deleting Elements:

- **Method** **Description**
  1. document.createElement(element) Create an HTML element
  2. document.removeChild(element) Remove an HTML element
  3. document.appendChild(element) Add an HTML element
  4. document.replaceChild(new, old) Replace an HTML element
  5. document.write(text) Write into the HTML output stream

**Example:**

```
<!DOCTYPE html>
<html>
<head>
  <title>DOM Manipulation Example</title>
</head>
<body>
  <h1>Shopping List</h1>
  <ul id="shoppingList">
    <li>Apples</li>
    <li>Bananas</li>
    <li>Oranges</li>
  </ul>
  <button onclick="addItem()">Add Item</button>
  <button onclick="deleteItem()">Delete Item</button>
  <script>
    // Function to add a new item to the list
    function addItem() {
      var shoppingList = document.getElementById("shoppingList");
      var newItem = document.createElement("li");
      newItem.textContent = "Grapes"; // Text for the new item
      shoppingList.appendChild(newItem);
    }
    // Function to delete an item from the list
    function deleteItem() {
      var shoppingList = document.getElementById("shoppingList");
      var items = shoppingList.getElementsByTagName("li");
      if (items.length > 0) {
        shoppingList.removeChild(items[0]); // Delete the first item
      }
    }
  </script>
</body>
</html>
```

### ❖ Adding Events Handlers:

- Method Description
  - document.getElementById(id).onclick = function(){code} Adding event handler code to an onclick event

# ❖ Finding HTML Objects:

- Finding HTML objects is a fundamental concept in web development that involves locating and interacting with specific elements within an HTML document using JavaScript.
- This concept is essential for creating dynamic and interactive web pages.

❖ **Document Object Model (DOM):** The DOM is a programming interface for web documents. It represents the structure of an HTML document as a tree of objects, where each object corresponds to an element in the HTML. JavaScript can be used to manipulate this tree, which allows you to find and interact with HTML elements.

❖ **Selecting Elements:** To find HTML objects within the DOM, you typically start by selecting or referencing them. There are various methods and techniques to select elements, including:

❖ **getElementById:** This method allows you to select an element by its unique id attribute.

❖ **getElementsByClassName:** You can select elements based on their class names. Multiple elements with the same class can be selected.

❖ **getElementsByTagName:** This method selects elements by their HTML tag name (e.g., "div," "p," "a").

❖ **querySelector and querySelectorAll:** These methods use CSS-style selectors to select elements. querySelector returns the first matching element, while querySelectorAll returns a collection of all matching elements.

❖ **Traversing the DOM:** You can navigate the DOM tree by moving from one element to another using properties like parentNode, childNodes, nextSibling, and previousSibling. This is useful for finding elements relative to a known reference point.

❖ **Using Properties and Methods:** Once you have a reference to an HTML element, you can use various properties and methods to interact with it. Common operations include changing the content, modifying attributes, altering the styling, attaching event listeners, and more.

❖ **Event Handling:** After finding HTML objects, you can attach event handlers to them. Event handlers allow you to respond to user interactions, such as clicks, mouse movements, and keyboard inputs, making your web page interactive.

**Example:**

```
<!DOCTYPE html>
<html>
<head>
  <title>Finding HTML Objects Example</title>
</head>
<body>
  <div id="myElementId">This is a div element with an id.</div>
  <script>
    // Find the element with id "myElementId"
    var element = document.getElementById("myElementId");
    // Modify the element's content
    element.textContent = "Updated content";
  </script>
</body>
</html>
```

**Note:** The first HTML DOM Level 1 (1998), defined 11 HTML objects, object collections, and properties. These are still valid in HTML5

| Property | Description | DOM |
|---|---|---|
| • document.anchors | Returns all <a> elements that have a name attribute | 1 |
| • document.applets | Deprecated | 1 |
| • document.baseURI | Returns the absolute base URI of the document | 3 |
| • document.body | Returns the <body> element | 1 |
| • document.cookie | Returns the document's cookie | 1 |
| • document.doctype | Returns the document's doctype | 3 |
| • document.documentElement | Returns the <html> element | 3 |
| • document.documentMode | Returns the mode used by the browser | 3 |
| • document.documentURI | Returns the URI of the document | 3 |
| • document.domain | Returns the domain name of the document server | 1 |
| • document.domConfig | Obsolete. | 3 |
| • document.embeds | Returns all <embed> elements | 3 |
| • document.forms | Returns all <form> elements | 1 |
| • document.head | Returns the <head> element | 3 |
| • document.images | Returns all <img> elements | 1 |
| • document.implementation | Returns the DOM implementation | 3 |
| • document.inputEncoding | Returns the document's encoding (character set) | 3 |
| • document.lastModified | Returns the date and time the document was updated | 3 |
| • document.links | Returns all <area> and <a> elements that have a href attribute | |
| • document.readyState | Returns the (loading) status of the document | 3 |
| • document.referrer | Returns the URI of the referrer (the linking document) | 1 |
| • document.scripts | Returns all <script> elements | 3 |
| • document.strictErrorChecking | Returns if error checking is enforced | 3 |
| • document.title | Returns the <title> element | 1 |
| • document.URL | Returns the complete URL of the document | 1 |

## Example:

```
<!DOCTYPE html>
<html>
<head>
  <title>DOM Properties and Methods Example</title>
</head>
<body>
  <a name="section1">Section 1</a>
  <a name="section2">Section 2</a>
  <script>
    // document.anchors - Returns all <a> elements that have a name attribute
    var anchors = document.anchors;
    console.log("Anchors with name attribute:");
    for (var i = 0; i < anchors.length; i++) {
      console.log(anchors[i]);
    }
    // document.baseURI - Returns the absolute base URI of the document
    var baseURI = document.baseURI;
    console.log("Base URI of the document:", baseURI);
```

```
    // document.body - Returns the <body> element
    var bodyElement = document.body;
    console.log("Body element:", bodyElement);
    // document.cookie - Returns the document's cookie
    var cookies = document.cookie;
    console.log("Document's cookie:", cookies);
    // document.doctype - Returns the document's doctype
    var doctype = document.doctype;
    console.log("Document's doctype:");
    console.log(doctype);
    // document.documentElement - Returns the <html> element
    var htmlElement = document.documentElement;
    console.log("HTML element:", htmlElement);
    // document.embeds - Returns all <embed> elements
    var embeds = document.embeds;
    console.log("Embed elements:");
    for (var i = 0; i < embeds.length; i++) {
      console.log(embeds[i]);
    }
    // document.head - Returns the <head> element
    var headElement = document.head;
    console.log("Head element:", headElement);
    // document.images - Returns all <img> elements
    var images = document.images;
    console.log("Image elements:");
    for (var i = 0; i < images.length; i++) {
      console.log(images[i]);
    }
    // document.title - Returns the <title> element
    var titleElement = document.title;
    console.log("Title element:", titleElement);
    // document.URL - Returns the URL of the document
    var documentURL = document.URL;
    console.log("Document URL:", documentURL);
  </script>
</body>
</html>
```

# JavaScript HTML DOM Elements:

- JavaScript HTML DOM (Document Object Model) elements are the representation of HTML elements within a web page, accessible and manipulable through JavaScript.

1. **Accessing DOM Elements:**
   - **getElementById:** This method allows you to select an element by its unique id attribute.
   - **getElementsByClassName:** Select elements based on their class names.
   - **getElementsByTagName:** Select elements by their HTML tag name (e.g., "div," "p," "a").
   - **querySelector and querySelectorAll:** Use CSS-style selectors to select elements.
   - **Traversing the DOM:** Navigate the DOM tree using properties like parentNode, childNodes, nextSibling, and previousSibling to find elements relative to a known reference point.
2. **Properties and Methods:**
   - **textContent:** Gets or sets the text content of an element.
   - **innerHTML:** Gets or sets the HTML content of an element.
   - **attributes:** Access and manipulate attributes of an element.
   - **style:** Access and manipulate CSS styles of an element.
   - **addEventListener:** Attach event listeners to respond to user interactions.
   - **removeEventListener:** Remove previously attached event listeners.
   - **appendChild and removeChild:** Add and remove child elements within parent elements.
   - **classList:** Access and manipulate the class attribute to add or remove CSS classes.
   - **getAttribute and setAttribute:** Get and set attributes of elements.
   - **value:** Access or change the value of form elements like input fields and textareas.
   - **parentNode and childNodes:** Access a node's parent and child elements.

## 3. Common DOM Element Properties:

- **tagName:** Returns the tag name of the element (e.g., "DIV" for a  element).
- **id:** Returns the id attribute of the element.
- **className:** Returns the value of the class attribute.
- **href:** Returns the href attribute for anchor elements (<a>).
- **src:** Returns the src attribute for image elements (<img>).
- **innerHTML:** Returns or sets the HTML content of the element.

## 4. Events:

- JavaScript can be used to attach event listeners to DOM elements to respond to user interactions such as clicks, mouse movements, keyboard inputs, etc.
- Common events include click, mouseover, keydown, submit, and more. Event listeners can be added using addEventListener() and removed using removeEventListener().

## 5. Modifying DOM Elements:

- JavaScript can dynamically modify the content, structure, and style of HTML elements in response to user actions or other events.
- Elements can be created, removed, replaced, and manipulated as needed.

6. **Security Considerations:**
   - Care should be taken when working with the DOM to prevent cross-site scripting (XSS) attacks.
   - Always sanitize and validate input and avoid injecting untrusted data into the DOM.
7. **Browser Compatibility:**
   - While the DOM is a standardized API, there can be variations in browser implementations. It's important to test and ensure cross-browser compatibility.

# JavaScript HTML DOM Events:

- JavaScript HTML DOM events are interactions and occurrences that happen in a web page, which can be detected and handled by JavaScript.
- Events can be triggered by user actions, such as mouse clicks, keyboard inputs, or changes in the document's state, and they provide a way to create dynamic and interactive web applications.

**Example:**

## 1. Event Types:

- **User Events:** These events are triggered by user interactions, such as mouse clicks (click), mouse movements (mousemove), keyboard inputs (keydown, keyup), and form submissions.
- **Document Events:** These events are related to the state of the document, such as when the document is fully loaded (DOMContentLoaded), when it is completely loaded (load), or when it is unloaded (unload).
- **Focus Events:** These events are triggered when an element receives or loses focus, such as focus and blur events.
- **Form Events:** Events like change, input, and submit are related to HTML form elements and user inputs.
- **Media Events:** Events like play, pause, and ended are related to audio and video elements.
- **Network Events:** Events like online and offline are related to the network connection status.
- **Custom Events:** Developers can create custom events to handle specific application logic.

## 2. Event Handling:

- Event handling is the process of defining JavaScript functions (event handlers or listeners) that respond to specific events.
- Event listeners are attached to HTML elements using methods like addEventListener() to specify which events to listen for and what function to call when the event occurs.

## 3. Event Object:

- When an event occurs, an event object is created, which contains information about the event, such as the type of event, the target element, mouse coordinates, and more.
- Event object properties and methods can be used to access and manipulate event data.

## 4. Event Propagation:

- Events in the DOM follow a propagation model, which determines the order in which elements receive and handle events.
- There are two phases of event propagation: capturing phase and bubbling phase.
- Event propagation can be controlled using the event.stopPropagation() and event.preventDefault() methods.

## 5. Event Delegation:

- Event delegation is a technique where a single event listener is placed on a common ancestor of multiple elements.
- It allows you to handle events for dynamically created or numerous elements efficiently, as events bubble up to the ancestor element.

## 6. Removing Event Listeners:

- Event listeners should be removed when they are no longer needed to prevent memory leaks.
- You can remove event listeners using the removeEventListener() method.

## 7. Browser Compatibility:

- While DOM events are standardized, there may be variations in event handling across different browsers.
- Feature detection and cross-browser testing are essential to ensure compatibility.

### 8. Event-driven Programming:

- JavaScript's event-driven nature is fundamental to modern web development and is used to create responsive and interactive user interfaces.

### 9. Common Use Cases:

- Form validation and submission
- UI interactions like button clicks and menu selections
- Animations and transitions
- Real-time updates and data synchronization
- Handling user inputs and providing feedback
- Implementing drag-and-drop functionality

## ❖ DOM events in JavaScript:

1. **Mouse Events:**
   - **click:** Triggered when the mouse button is clicked.
   - **mousedown:** Triggered when the mouse button is pressed down.
   - **mouseup:** Triggered when the mouse button is released.
   - **mousemove:** Triggered when the mouse pointer moves.
   - **mouseover:** Triggered when the mouse pointer enters an element.
   - **mouseout:** Triggered when the mouse pointer leaves an element.
   - **dblclick:** Triggered when the mouse button is double-clicked.
2. **Keyboard Events:**
   - **keydown:** Triggered when a keyboard key is pressed down.
   - **keyup:** Triggered when a keyboard key is released.
   - **keypress:** Triggered when a character key is pressed down and released.
3. **Form Events:**
   - **submit:** Triggered when a form is submitted.
   - **reset:** Triggered when a form is reset.
   - **change:** Triggered when the value of a form element (input, select, etc.) changes.
   - **input:** Triggered when the value of an input element changes (more responsive than change).
   - **focus:** Triggered when an element receives focus.
   - **blur:** Triggered when an element loses focus.
   - **select:** Triggered when text is selected within an input or textarea element.
4. **Window and Document Events:**
   - **load:** Triggered when the document or a resource (e.g., image) finishes loading.
   - **unload:** Triggered when the user navigates away from the page.
   - **resize:** Triggered when the browser window is resized.
   - **scroll:** Triggered when the user scrolls within an element.
   - **DOMContentLoaded:** Triggered when the HTML document is fully loaded and parsed.
   - **beforeunload:** Triggered before the user leaves the page (for confirmation).
5. **Drag-and-Drop Events:**
   - **dragstart:** Triggered when an element is dragged.
   - **dragend:** Triggered when the element's drag operation is completed.

- **dragover:** Triggered when an element is being dragged over a valid drop target.
- **drop:** Triggered when an element is dropped onto a valid drop target.

6. **Media Events:**
   - **play:** Triggered when media (audio/video) starts playing.
   - **pause:** Triggered when media is paused.
   - **ended:** Triggered when media playback reaches the end.
7. **Network Events:**
   - **online:** Triggered when the browser detects an internet connection.
   - **offline:** Triggered when the browser loses its internet connection.
8. **Custom Events:**
   - Developers can create and dispatch custom events using the CustomEvent constructor.
   - Touch Events (for mobile and touch-enabled devices):
   - touchstart: Triggered when a touch point is placed on the screen.
   - touchmove: Triggered when a touch point moves along the screen.
   - touchend: Triggered when a touch point is removed from the screen.

**Example:**

❖ **Mouse Events:**

- onclick or click:
- mousedown:
- mouseup:
- mousemove:
- mouseover:
- mouseover:
- mouseout:
- dblclick:

➢ **raj.html**

```
<!DOCTYPE html>
<html>
<head>
  <title>Mouse Events Example</title>
  <style>
    #myElement {
      width: 100px;
      height: 100px;
      background-color: lightblue;
      text-align: center;
      line-height: 100px;
      cursor: pointer;
    }
  </style>
</head>
<body>
  <div id="myElement">Click Me</div>
  <script>
    var myElement = document.getElementById("myElement");
    // Click event
```

```
        myElement.onclick = function () {
          myElement.style.backgroundColor = "lightgreen";
          myElement.textContent = "Clicked!";
        };
        // Mousedown event
        myElement.onmousedown = function () {
          myElement.style.backgroundColor = "lightcoral";
          myElement.textContent = "Mouse Down";
        };
        // Mouseup event
        myElement.onmouseup = function () {
          myElement.style.backgroundColor = "lightblue";
          myElement.textContent = "Mouse Up";
        };
        // Mousemove event
        myElement.onmousemove = function () {
          myElement.style.backgroundColor = "lightpink";
        };
        // Mouseover event
        myElement.onmouseover = function () {
          myElement.style.border = "2px solid red";
        };
        // Mouseout event
        myElement.onmouseout = function () {
          myElement.style.border = "none";
        };
        // Double click event
        myElement.ondblclick = function () {
          myElement.style.backgroundColor = "lightyellow";
          myElement.textContent = "Double Clicked!";
        };
      </script>
    </body>
    </html>
```

## Example:

- **keyword events:**
  - **Raj.html**

```
<!DOCTYPE html>
<html>
<head>
  <title>Keyboard Events Example</title>
</head>
<body>
  <h1>Keyboard Events Example</h1>
  <p>Press a key on your keyboard to see the events in action.</p>
  <script>
    // Function to handle keydown event
```

```
        function handleKeyDown(event) {
          const key = event.key;
          console.log(`Keydown Event: Key pressed - ${key}`);
        }
        // Function to handle keyup event
        function handleKeyUp(event) {
          const key = event.key;
          console.log(`Keyup Event: Key released - ${key}`);
        }
        // Function to handle keypress event
        function handleKeyPress(event) {
          const charCode = event.charCode;
          console.log(`Keypress Event: Character code - ${charCode}`);
        }
        // Add event listeners for keydown, keyup, and keypress events
        document.addEventListener('keydown', handleKeyDown);
        document.addEventListener('keyup', handleKeyUp);
        document.addEventListener('keypress', handleKeyPress);
      </script>
    </body>
    </html>
```

**Example:**

- **Form Events:**
- **Example-1**
  - **Raj.html**

```
<!DOCTYPE html>
<html>
<head>
  <title>Form Events Example</title>
</head>
<body>
  <h1>Form Events Example</h1>
  <form id="exampleForm">
    <label for="inputField">Input Field:</label>
    <input type="text" id="inputField" name="inputField">
    <br><br>
    <label for="selectField">Select Field:</label>
    <select id="selectField" name="selectField">
      <option value="option1">Option 1</option>
      <option value="option2">Option 2</option>
      <option value="option3">Option 3</option>
    </select>
    <br><br>
    <input type="submit" value="Submit">
    <input type="reset" value="Reset">
  </form>
  <script>
```

```
    // Function to handle submit event
    function handleSubmit(event) {
      event.preventDefault();
      console.log("Submit Event: Form submitted");
    }
    // Function to handle reset event
    function handleReset(event) {
      console.log("Reset Event: Form reset");
    }
    // Function to handle change event (select field)
    function handleChange(event) {
      const selectedValue = event.target.value;
      console.log(`Change Event: Selected value - ${selectedValue}`);
    }
    // Function to handle input event (text input field)
    function handleInput(event) {
      const inputValue = event.target.value;
      console.log(`Input Event: Input value - ${inputValue}`);
    }
    // Function to handle focus event (text input field)
    function handleFocus(event) {
      console.log("Focus Event: Input field focused");
    }
    // Function to handle blur event (text input field)
    function handleBlur(event) {
      console.log("Blur Event: Input field blurred");
    }
    // Function to handle select event (select field)
    function handleSelect(event) {
      console.log("Select Event: Text selected");
    }
    // Add event listeners for form events
    const form = document.getElementById('exampleForm');
    form.addEventListener('submit', handleSubmit);
    form.addEventListener('reset', handleReset);
    const selectField = document.getElementById('selectField');
    selectField.addEventListener('change', handleChange);
    const inputField = document.getElementById('inputField');
    inputField.addEventListener('input', handleInput);
    inputField.addEventListener('focus', handleFocus);
    inputField.addEventListener('blur', handleBlur);
    inputField.addEventListener('select', handleSelect);
  </script>
</body>
</html>
```

**Example-2:**

```
<!DOCTYPE html>
<html>
<head>
  <title>Form Events Example</title>
</head>
<body>
  <h1>Form Events Example</h1>
  <form id="exampleForm">
    <label for="inputField">Input Field:</label>
    <input type="text" id="inputField" name="inputField">
    <br><br>
    <label for="selectField">Select Field:</label>
    <select id="selectField" name="selectField">
      <option value="option1">Option 1</option>
      <option value="option2">Option 2</option>
      <option value="option3">Option 3</option>
    </select>
    <br><br>
    <input type="submit" value="Submit">
    <input type="reset" value="Reset">
  </form>
  <div id="output"></div>
  <script>
    // Function to handle submit event
    function handleSubmit(event) {
      event.preventDefault();
      document.getElementById('output').innerText = "Submit Event: Form submitted";
    }
    // Function to handle reset event
    function handleReset(event) {
      document.getElementById('output').innerText = "Reset Event: Form reset";
    }
    // Function to handle change event (select field)
    function handleChange(event) {
      const selectedValue = event.target.value;
      document.getElementById('output').innerText = `Change Event: Selected value - ${selectedValue}`;
    }
    // Function to handle input event (text input field)
    function handleInput(event) {
      const inputValue = event.target.value;
      document.getElementById('output').innerText = `Input Event: Input value - ${inputValue}`;
    }
    // Function to handle focus event (text input field)
    function handleFocus(event) {
```

```
        document.getElementById('output').innerText = "Focus Event: Input field focused";
      }
      // Function to handle blur event (text input field)
      function handleBlur(event) {
        document.getElementById('output').innerText = "Blur Event: Input field blurred";
      }
      // Function to handle select event (select field)
      function handleSelect(event) {
        document.getElementById('output').innerText = "Select Event: Text selected";
      }
      // Add event listeners for form events
      const form = document.getElementById('exampleForm');
      form.addEventListener('submit', handleSubmit);
      form.addEventListener('reset', handleReset);
      const selectField = document.getElementById('selectField');
      selectField.addEventListener('change', handleChange);
      const inputField = document.getElementById('inputField');
      inputField.addEventListener('input', handleInput);
      inputField.addEventListener('focus', handleFocus);
      inputField.addEventListener('blur', handleBlur);
      inputField.addEventListener('select', handleSelect);
    </script>
  </body>
  </html>
```

**Example:-**

- ❖ Window and Document Events:
  - load:
  - unload:
  - resize:
  - scroll:
  - DOMContentLoaded:
  - beforeunload:

➢ **raj.html:**

```
<!DOCTYPE html>
<html>
<head>
  <title>Window and Document Events Example</title>
</head>
<body>
  <h1>Window and Document Events Example</h1>
  <p>Scroll down to see the "scroll" event in action.</p>
  <script>
    // Function to handle the "load" event for the window
    window.addEventListener('load', function() {
      console.log('Window Load Event: The page has fully loaded.');
    });
    // Function to handle the "unload" event for the window
```

```
        window.addEventListener('unload', function() {
          console.log('Window Unload Event: The page is about to unload.');
        });
        // Function to handle the "resize" event for the window
        window.addEventListener('resize', function() {
          console.log('Window Resize Event: The window size has changed.');
        });
        // Function to handle the "scroll" event for the window
        window.addEventListener('scroll', function() {
          console.log('Window Scroll Event: Scrolling detected.');
        });
        // Function to handle the "DOMContentLoaded" event for the document
        document.addEventListener('DOMContentLoaded', function() {
          console.log('DOMContentLoaded Event: The HTML document has been fully loaded and
parsed.');
        });
        // Function to handle the "beforeunload" event for the window
        window.addEventListener('beforeunload', function(event) {
          event.preventDefault();
          event.returnValue = '';
          console.log('Window Beforeunload Event: The page is about to be unloaded.');
        });
      </script>
    </body>
    </html>
```

**Example:**

❖ **Drag-and-Drop Events:**

- dragstart:
- dragend:
- dragover:
- drop:

➢ **raj.html:**

```
<!DOCTYPE html>
<html>
<head>
  <title>Drag-and-Drop Events Example</title>
  <style>
    #drag-source {
      width: 100px;
      height: 100px;
      background-color: lightblue;
      text-align: center;
      padding: 10px;
      cursor: pointer;
    }
    #drop-target {
      width: 200px;
```

```
      height: 200px;
      border: 2px dashed gray;
      margin-top: 20px;
    }
  </style>
</head>
<body>
  <h1>Drag-and-Drop Events Example</h1>
  <div id="drag-source" draggable="true">
    Drag Me!
  </div>
  <div id="drop-target">
    Drop Here!
  </div>
  <script>
    const dragSource = document.getElementById('drag-source');
    const dropTarget = document.getElementById('drop-target');
    // Function to handle the "dragstart" event
    dragSource.addEventListener('dragstart', function (event) {
      event.dataTransfer.setData('text/plain', 'Dragged element');
      console.log('Drag Start Event: The drag operation has started.');
    });
    // Function to handle the "dragend" event
    dragSource.addEventListener('dragend', function () {
      console.log('Drag End Event: The drag operation has ended.');
    });
    // Function to handle the "dragover" event
    dropTarget.addEventListener('dragover', function (event) {
      event.preventDefault();
      console.log('Drag Over Event: Element is being dragged over the drop target.');
    });
    // Function to handle the "drop" event
    dropTarget.addEventListener('drop', function (event) {
      event.preventDefault();
      const data = event.dataTransfer.getData('text/plain');
      dropTarget.innerText = `Dropped: ${data}`;
      console.log('Drop Event: Element has been dropped on the target.');
    });
  </script>
</body>
</html>
```

## Example:

❖ **Media Events:**

- play:
- pause:
- ended:

➢ **raj.html:**

```
<!DOCTYPE html>
<html>
<head>
  <title>Media Events Example</title>
  <style>
    #video-container {
      text-align: center;
    }
    video {
      max-width: 100%;
    }
    #play-button {
      background-color: #4CAF50;
      color: white;
      border: none;
      padding: 10px 20px;
      font-size: 16px;
      cursor: pointer;
      margin-right: 10px;
    }
  </style>
</head>
<body>
  <h1>Media Events Example</h1>
  <div id="video-container">
    <video id="myVideo" controls>
      <source src="sample.mp4" type="video/mp4">
      Your browser does not support the video tag.
    </video>
    <button id="play-button">Play</button>
  </div>
  <script>
    const video = document.getElementById('myVideo');
    const playButton = document.getElementById('play-button');
    // Function to handle the "play" event
    video.addEventListener('play', function () {
      console.log('Play Event: Video playback has started.');
    });
    // Function to handle the "pause" event
    video.addEventListener('pause', function () {
```

```
    console.log('Pause Event: Video playback has been paused.');
  });
  // Function to handle the "ended" event
  video.addEventListener('ended', function () {
    console.log('Ended Event: Video playback has ended.');
  });
  // Function to handle the play button click event
  playButton.addEventListener('click', function () {
    if (video.paused) {
      video.play();
      playButton.textContent = 'Pause';
    } else {
      video.pause();
      playButton.textContent = 'Play';
    }
  });
 </script>
</body>
</html>
```

## Example:

- **Network Events:**
  - online:
  - offline:

- **raj.html**

```
<!DOCTYPE html>
<html>
<head>
  <title>Network Events Example</title>
  <style>
    body {
      text-align: center;
    }
    #status {
      font-size: 24px;
      margin-top: 50px;
    }
  </style>
</head>
<body>
  <h1>Network Events Example</h1>
  <p>This page will display your network status.</p>
  <div id="status">Loading...</div>
  <script>
    const statusElement = document.getElementById('status');
    // Function to update the network status
    function updateNetworkStatus() {
      if (navigator.onLine) {
```

```
            statusElement.textContent = 'You are online!';
            statusElement.style.color = 'green';
          } else {
            statusElement.textContent = 'You are offline.';
            statusElement.style.color = 'red';
          }
        }
        // Event listener for the "online" event
        window.addEventListener('online', function () {
          updateNetworkStatus();
          console.log('Online Event: You are now online.');
        });
        // Event listener for the "offline" event
        window.addEventListener('offline', function () {
          updateNetworkStatus();
          console.log('Offline Event: You are now offline.');
        });
        // Initial check and update of network status
        updateNetworkStatus();
      </script>
    </body>
    </html>
```

Example:

- Custom Events:
  - Developers can create and dispatch custom events using the CustomEvent constructor.
  - Touch Events (for mobile and touch-enabled devices):
  - touchstart:
  - touchmove:
  - touchend:

- **raj.html:**

```
<!DOCTYPE html>
<html>
<head>
  <title>Custom Events and Touch Events Example</title>
  <style>
    /* Add your CSS styles here */
    body {
      font-family: Arial, sans-serif;
      text-align: center;
    }
    #custom-event-section {
      padding: 20px;
      border: 1px solid #ccc;
      margin-bottom: 20px;
    }
    #touch-event-section {
      padding: 20px;
```

```
        border: 1px solid #ccc;
        margin-bottom: 20px;
      }
    </style>
</head>
<body>
    <h1>Custom Events and Touch Events Example</h1>
    <!-- Custom Events Example -->
    <div id="custom-event-section">
      <h2>Custom Events Example</h2>
      <button id="custom-event-button">Click Me</button>
      <div id="custom-event-output"></div>
    </div>
    <!-- Touch Events Example -->
    <div id="touch-event-section">
      <h2>Touch Events Example</h2>
      <div id="touch-event-output"></div>
    </div>
    <script>
      // Custom Events Example
      const customEventButton = document.getElementById('custom-event-button');
      const customEventOutput = document.getElementById('custom-event-output');
      // Create a custom event
      const customEvent = new Event('customEvent');
      customEventButton.addEventListener('click', function () {
        // Dispatch the custom event when the button is clicked
        customEventOutput.innerText = 'Custom Event: Button clicked!';
        customEventButton.dispatchEvent(customEvent);
      });
      // Listen for the custom event and handle it
      customEventButton.addEventListener('customEvent', function () {
        customEventOutput.innerText = 'Custom Event: Custom event triggered!';
      });
      // Touch Events Example
      const touchEventOutput = document.getElementById('touch-event-output');
      // Function to handle touchstart event
      function handleTouchStart(event) {
        touchEventOutput.innerText = 'Touch Start Event: Touch detected.';
      }

      // Function to handle touchmove event
      function handleTouchMove(event) {
        touchEventOutput.innerText = 'Touch Move Event: Touch moving.';
      }
      // Function to handle touchend event
      function handleTouchEnd(event) {
        touchEventOutput.innerText = 'Touch End Event: Touch ended.';
      }
```

```
        // Add touch event listeners
        touchEventOutput.addEventListener('touchstart', handleTouchStart);
        touchEventOutput.addEventListener('touchmove', handleTouchMove);
        touchEventOutput.addEventListener('touchend', handleTouchEnd);
    </script>
</body>
</html>
```

## Complete Example event in JavaSrcipt with HTML and CSS.

- **Raj.html**

```
<!DOCTYPE html>
<html>
<head>
    <title>All Events Example</title>
    <style>
        /* Add your CSS styles here */
        body {
            font-family: Arial, sans-serif;
        }
        #keyboard-events {
            padding: 20px;
            border: 1px solid #ccc;
            margin-bottom: 20px;
        }
        #form-events {
            padding: 20px;
            border: 1px solid #ccc;
            margin-bottom: 20px;
        }
        #media-events {
            padding: 20px;
            border: 1px solid #ccc;
            margin-bottom: 20px;
        }
        #network-events {
            padding: 20px;
            border: 1px solid #ccc;
        }
    </style>
</head>
<body>
    <h1>All Events Example</h1>
    <!-- Keyboard Events Example -->
    <div id="keyboard-events">
        <h2>Keyboard Events Example</h2>
        <input type="text" id="keyboard-input">
        <div id="keyboard-output"></div>
    </div>
```

```
<!-- Form Events Example -->
<div id="form-events">
  <h2>Form Events Example</h2>
  <form id="exampleForm">
    <label for="inputField">Input Field:</label>
    <input type="text" id="inputField" name="inputField">
    <br><br>
    <label for="selectField">Select Field:</label>
    <select id="selectField" name="selectField">
      <option value="option1">Option 1</option>
      <option value="option2">Option 2</option>
      <option value="option3">Option 3</option>
    </select>
    <br><br>
    <input type="submit" value="Submit">
    <input type="reset" value="Reset">
  </form>
  <div id="form-output"></div>
</div>
<!-- Media Events Example -->
<div id="media-events">
  <h2>Media Events Example</h2>
  <video id="myVideo" controls>
    <source src="sample.mp4" type="video/mp4">
    Your browser does not support the video tag.
  </video>
  <button id="play-button">Play</button>
  <div id="media-output"></div>
</div>
<!-- Network Events Example -->
<div id="network-events">
  <h2>Network Events Example</h2>
  <div id="network-status">Loading...</div>
</div>
<script>
  // Keyboard Events Example
  const keyboardInput = document.getElementById('keyboard-input');
  const keyboardOutput = document.getElementById('keyboard-output');
  keyboardInput.addEventListener('keydown', function (event) {
    const key = event.key;
    keyboardOutput.innerText = `Keydown Event: Key pressed - ${key}`;
  });
  // Form Events Example
  const form = document.getElementById('exampleForm');
  const formOutput = document.getElementById('form-output');
  form.addEventListener('submit', function (event) {
    event.preventDefault();
    formOutput.innerText = 'Submit Event: Form submitted';
```

```
    });
    form.addEventListener('reset', function () {
      formOutput.innerText = 'Reset Event: Form reset';
    });
    // Media Events Example
    const video = document.getElementById('myVideo');
    const playButton = document.getElementById('play-button');
    const mediaOutput = document.getElementById('media-output');
    video.addEventListener('play', function () {
      mediaOutput.innerText = 'Play Event: Video playback has started.';
    });
    video.addEventListener('pause', function () {
      mediaOutput.innerText = 'Pause Event: Video playback has been paused.';
    });
    video.addEventListener('ended', function () {
      mediaOutput.innerText = 'Ended Event: Video playback has ended.';
    });
    playButton.addEventListener('click', function () {
      if (video.paused) {
        video.play();
        playButton.textContent = 'Pause';
      } else {
        video.pause();
        playButton.textContent = 'Play';
      }
    });
    // Network Events Example
    const networkStatus = document.getElementById('network-status');
    function updateNetworkStatus() {
      if (navigator.onLine) {
        networkStatus.innerText = 'Online Event: You are now online.';
        networkStatus.style.color = 'green';
      } else {
        networkStatus.innerText = 'Offline Event: You are now offline.';
        networkStatus.style.color = 'red';
      }
    }
    window.addEventListener('online', updateNetworkStatus);
    window.addEventListener('offline', updateNetworkStatus);
    updateNetworkStatus();
  </script>
</body>
</html>
```

# Regular expressions in JavaScript

**1. Matching Digits:**
- Match any digit: `\d`
- Match any non-digit: `\D`

**2. Matching Words and Characters:**
- Match any word character (alphanumeric plus underscore): `\w`
- Match any non-word character: `\W`

**3. Matching Whitespace:**
- Match any whitespace character: `\s`
- Match any non-whitespace character: `\S`

**4. Matching Specific Characters:**
- Match a specific character (e.g., 'a'): `a`
- Match a character range (e.g., 'a' to 'z'): `[a-z]`
- Match any character except those in a range: `[^0-9]` (matches anything that is not a digit)

**5. Quantifiers (Repetition):**
- Match zero or more: `*`
- Match one or more: `+`
- Match zero or one: `?`
- Match exactly n times: `{n}`
- Match n or more times: `{n,}`
- Match between n and m times: `{n,m}`

**6. Anchors:**
- Match the start of a line or string: `^`
- Match the end of a line or string: `$`

**7. Grouping and Capturing:**
- Create a capturing group: `(pattern)`
- Refer to captured groups: `\1`, `\2`, etc.

**8. Alternation:**
- Match one of several patterns: `pattern1|pattern2`

**9. Word Boundaries:**
- Match at the beginning of a word: `\b`
- Match at the end of a word: `\b`

**10. Quantifier Shortcuts:**
- Match exactly one digit: `\d{1}`
- Match two digits: `\d{2}`
- Match between 3 and 5 word characters: `\w{3,5}`

**11. Email Validation:**
- Basic email validation (not exhaustive): `^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`

**12. URL Validation:**
- Basic URL validation (not exhaustive): `^(https?|ftp)://[^\s/$.?#].[^\s]*$`

**13. IP Address Validation:**
- IPv4 address validation (not exhaustive): `^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$`

**Example-1:**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Regular Expression Validation</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }
    label {
      display: block;
      margin-bottom: 5px;
    }
    input[type="text"] {
      width: 100%;
      padding: 8px;
      margin-bottom: 10px;
      border: 1px solid #ccc;
    }
    input[type="submit"] {
      background-color: #007BFF;
      color: white;
      border: none;
      padding: 10px 20px;
      cursor: pointer;
    }
  </style>
</head>
<body>
  <h1>Regular Expression Validation</h1>
  <form id="validationForm">
    <label for="email">Email:</label>
    <input type="text" id="email" name="email" required pattern="^[a-zA-Z0-9._%+-]+@[a-
zA-Z0-9.-]+\.[a-zA-Z]{2,}$">
    <label for="password">Password (at least 8 characters):</label>
    <input type="password" id="password" name="password" required pattern=".{8,}">
    <label for="date">Date (yyyy-mm-dd):</label>
    <input type="text" id="date" name="date" required pattern="^\d{4}-\d{2}-\d{2}$">
    <input type="submit" value="Submit">
  </form>
  <script>
    const form = document.getElementById('validationForm');
    form.addEventListener('submit', function (event) {
      const emailInput = document.getElementById('email');
      const passwordInput = document.getElementById('password');
      const dateInput = document.getElementById('date');
```

```
        if (!emailInput.checkValidity()) {
          alert('Invalid email address');
          event.preventDefault();
        }
        if (!passwordInput.checkValidity()) {
          alert('Password must be at least 8 characters long');
          event.preventDefault();
        }
        if (!dateInput.checkValidity()) {
          alert('Invalid date format (yyyy-mm-dd)');
          event.preventDefault();
        }
      });
    </script>
  </body>
  </html>
```

**Example-2:**

- **Raj.html**

```
<!DOCTYPE html>
<html>
<head>
  <title>Regular Expression Example</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }
    input {
      width: 100%;
      padding: 10px;
      margin-bottom: 10px;
    }
    button {
      padding: 10px 20px;
      font-size: 16px;
    }
    #result {
      margin-top: 20px;
      color: green;
    }
    .error {
      color: red;
    }
  </style>
</head>
<body>
  <h1>Regular Expression Example</h1>
```

```
    <label for="email">Email:</label>
    <input type="text" id="email" placeholder="Enter email">
    <br>
    <label for="password">Password:</label>
    <input type="password" id="password" placeholder="Enter password">
    <br>
    <label for="ip">IP Address:</label>
    <input type="text" id="ip" placeholder="Enter IP address">
    <br>
    <button onclick="validateInput()">Validate</button>
    <div id="result"></div>
    <script src="raj.js"></script>
  </body>
  </html>
```

- **Raj.js:**

```
function validateInput() {
  const email = document.getElementById('email').value;
  const password = document.getElementById('password').value;
  const ip = document.getElementById('ip').value;
  const emailPattern = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
  const passwordPattern = /^(?=.*[A-Za-z])(?=.*\d)[A-Za-z\d]{8,}$/;
  const ipPattern = /^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-
9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$/;
  const result = document.getElementById('result');
  result.innerHTML = '';
  if (!emailPattern.test(email)) {
    result.innerHTML += '<div class="error">Invalid email address</div>';
  }
  if (!passwordPattern.test(password)) {
    result.innerHTML += '<div class="error">Invalid password. Must contain at least 8
characters with at least one letter and one number.</div>';
  }
  if (!ipPattern.test(ip)) {
    result.innerHTML += '<div class="error">Invalid IP address</div>';
  }
  if (result.innerHTML === '') {
    result.innerHTML = '<div class="success">All inputs are valid!</div>';
  }
}
```

# JavaScript Validation

- Form Validation
- Email Validation

## ❖ Example of Form Validation:

➢ **index.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Form Validation Example</title>
  <link rel="stylesheet" href="Raj.css">
</head>
<body>
  <div class="container">
    <h1>Registration Form</h1>
    <form id="registrationForm">
      <div class="form-group">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required>
        <span class="error" id="nameError"></span>
      </div>
      <div class="form-group">
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required>
        <span class="error" id="emailError"></span>
      </div>
      <div class="form-group">
        <label for="password">Password:</label>
        <input type="password" id="password" name="password" required>
        <span class="error" id="passwordError"></span>
      </div>
      <div class="form-group">
        <label for="confirmPassword">Confirm Password:</label>
        <input type="password" id="confirmPassword" name="confirmPassword" required>
        <span class="error" id="confirmPasswordError"></span>
      </div>
      <button type="submit">Register</button>
    </form>
  </div>
  <script src="Raj.js"></script>
</body>
</html>
```

## ➢ Raj.js

```
document.addEventListener('DOMContentLoaded', function () {
  const registrationForm = document.getElementById('registrationForm');
  const nameInput = document.getElementById('name');
  const emailInput = document.getElementById('email');
  const passwordInput = document.getElementById('password');
  const confirmPasswordInput = document.getElementById('confirmPassword');
  const nameError = document.getElementById('nameError');
  const emailError = document.getElementById('emailError');
  const passwordError = document.getElementById('passwordError');
  const confirmPasswordError = document.getElementById('confirmPasswordError');
  registrationForm.addEventListener('submit', function (event) {
    let isValid = true;
    // Reset error messages
    nameError.textContent = '';
    emailError.textContent = '';
    passwordError.textContent = '';
    confirmPasswordError.textContent = '';
    // Name validation (letters and spaces only)
    const namePattern = /^[A-Za-z\s]+$/;
    if (!namePattern.test(nameInput.value)) {
      nameError.textContent = 'Name can only contain letters and spaces.';
      isValid = false;
    }
    // Email validation
    const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    if (!emailPattern.test(emailInput.value)) {
      emailError.textContent = 'Invalid email address.';
      isValid = false;
    }
    // Password validation (at least 8 characters)
    if (passwordInput.value.length < 8) {
      passwordError.textContent = 'Password must be at least 8 characters long.';
      isValid = false;
    }
    // Confirm password validation
    if (passwordInput.value !== confirmPasswordInput.value) {
      confirmPasswordError.textContent = 'Passwords do not match.';
      isValid = false;
    }
    if (!isValid) {
      event.preventDefault(); // Prevent form submission if there are errors
    }
  });
});
```

## ➢ Raj.css

```
body {
   font-family: Arial, sans-serif;
   background-color: #f2f2f2;
   margin: 0;
   padding: 0;
}
.container {
   background-color: #fff;
   max-width: 400px;
   margin: 0 auto;
   padding: 20px;
   border-radius: 5px;
   box-shadow: 0 0 5px rgba(0, 0, 0, 0.2);
}
h1 {
   text-align: center;
}
.form-group {
   margin-bottom: 20px;
}
label {
   display: block;
   font-weight: bold;
}
input[type="text"],
input[type="email"],
input[type="password"] {
   width: 100%;
   padding: 10px;
   border: 1px solid #ccc;
   border-radius: 3px;
}
.error {
   color: red;
   font-size: 0.9em;
}
button {
   background-color: #007bff;
   color: #fff;
   padding: 10px 20px;
   border: none;
   border-radius: 3px;
   cursor: pointer;
}
button:hover {
   background-color: #0056b3;}
```

## ❖ Example of Email Validation:

### ➢ index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Email Validation Example</title>
  <link rel="stylesheet" href="Raj.css">
</head>
<body>
  <div class="container">
    <h1>Email Validation</h1>
    <form id="emailValidationForm" onsubmit="return validateEmail()">
      <div class="form-group">
        <label for="email">Email:</label>
        <input type="email" id="email" required>
        <span class="error" id="emailError"></span>
      </div>
      <button type="submit">Validate</button>
    </form>
  </div>
  <script src="Raj.js"></script>
</body>
</html>
```

➢ Raj.js

```
function validateEmail() {
  const emailInput = document.getElementById('email');
  const emailError = document.getElementById('emailError');
  // Regular expression for basic email validation
  const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  if (!emailPattern.test(emailInput.value)) {
    emailError.textContent = 'Invalid email address.';
    return false; // Prevent form submission
  }
  // Clear any previous error message
  emailError.textContent = '';
  return true; // Allow form submission
}
```

- **Raj.css**

```
body {
   font-family: Arial, sans-serif;
   background-color: #f2f2f2;
   margin: 0;
   padding: 0;
}

.container {
   background-color: #fff;
   max-width: 400px;
   margin: 0 auto;
   padding: 20px;
   border-radius: 5px;
   box-shadow: 0 0 5px rgba(0, 0, 0, 0.2);
}
h1 {
   text-align: center;
}
.form-group {
   margin-bottom: 20px;
}
label {
   display: block;
   font-weight: bold;
}
input[type="email"] {
   width: 100%;
   padding: 10px;
   border: 1px solid #ccc;
   border-radius: 3px;
}
.error {
   color: red;
   font-size: 0.9em;
}
button {
   background-color: #007bff;
   color: #fff;
   padding: 10px 20px;
   border: none;
   border-radius: 3px;
   cursor: pointer;
}
button:hover {
   background-color: #0056b3;
}
```

# Web APIs

❖ **What is Web API?:**

- API stands for Application Programming Interface.
- A Web API is an application programming interface for the Web.
- A Browser API can extend the functionality of a web browser.
- A Server API can extend the functionality of a web server.

❖ **Browser APIs:**

- All browsers have a set of built-in Web APIs to support complex operations, and to help accessing data.
- For example, the Geolocation API can return the coordinates of where the browser is located.

**Example:**

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Geolocation</h2>
<p>Click the button to get your coordinates.</p>
<button onclick="getLocation()">Try It</button>
<p id="demo"></p>
<script>
const x = document.getElementById("demo");
function getLocation() {
  try {
    navigator.geolocation.getCurrentPosition(showPosition);
  } catch {
    x.innerHTML = err;
  }
}
function showPosition(position) {
  x.innerHTML = "Latitude: " + position.coords.latitude +
  "<br>Longitude: " + position.coords.longitude;
}
</script>
</body>
</html>
```

❖ **Third Party APIs:**

- Third party APIs are not built into your browser.
- To use these APIs, you will have to download the code from the Web.

**Examples:**

- YouTube API - Allows you to display videos on a web site.
- Twitter API - Allows you to display Tweets on a web site.
- Facebook API - Allows you to display Facebook info on a web site.

# ❖ JavaScript Validation API:

- The JavaScript Validation API, also known as the Constraint Validation API, is a set of built-in browser functions and properties that enable client-side form validation in web applications. It provides a convenient way to validate user input in HTML forms before submitting them to the server. The Validation API is available in modern web browsers and can be accessed using JavaScript.

❖ Features and components of the JavaScript Validation API

**1. HTML5 Validation Attributes:** HTML5 introduced several new attributes that can be added to form elements to specify validation rules. These attributes include `required`, `min`, `max`, `pattern`, and more. They are used to define constraints on form fields.

```
<input type="text" id="username" name="username" required minlength="3" maxlength="20">
```

**2. Constraint Validation Methods:** The Validation API provides methods that can be called on form elements to check their validity. The most commonly used methods include:

- `checkValidity()`: Checks if the input field's value meets all defined constraints and returns a Boolean (`true` if valid, `false` if not).
- `setCustomValidity(message)`: Allows you to set a custom validation message that is displayed when the `checkValidity()` method returns `false`.
- `reportValidity()`: Displays the validation message if the input is invalid, or returns `true` if the input is valid.

**3.Validity Properties:** Each form element has a set of validity properties that can be accessed to check its validation state. Common validity properties include:

- `validity.valid`: A Boolean indicating whether the field's value is valid.
- `validity.valueMissing`: A Boolean indicating whether the field is required and its value is empty.
- `validity.tooShort` and `validity.tooLong`: Booleans indicating whether the input length is less than or greater than the specified `minlength` or `maxlength`.

**4. Validation Messages:** When an input field is invalid, the Validation API displays a default validation message. However, you can customize these messages using the `setCustomValidity()` method or by defining the `title` attribute on the input element.

**5. Styling for Invalid Fields:** Invalid form elements can be styled using the `:invalid` CSS pseudo-class. This allows you to apply specific styles to invalid fields, such as changing the border color to red.

```
input:invalid {
  border-color: red;
}
```

**6. Form Submission Prevention:** When using Validation API, if a form is invalid (e.g., required fields are empty), it prevents the form from being submitted, giving users a chance to correct their input

**7. Event Handling:** You can listen for form-related events, such as `submit`, and use JavaScript to check form validity and prevent submission if necessary. Additionally, you can listen for events like `input` and `change` on form fields to provide real-time feedback to users as they fill out the form.

```
const form = document.getElementById('myForm');
form.addEventListener('submit', function(event) {
  if (!form.checkValidity()) {
    event.preventDefault(); // Prevent form submission
    // Display error messages or take other actions
  }
});
```

# JS AJAX

- AJAX (Asynchronous JavaScript and XML)** is a set of web development techniques used to create asynchronous web applications. It allows you to exchange data with a web server without having to reload the entire web page. AJAX leverages a combination of technologies, including JavaScript, XML (though often replaced with JSON), and the XMLHttpRequest object (or newer alternatives like Fetch API).

**1. XMLHttpRequest Object:**

- The `XMLHttpRequest` object is a key component of AJAX. It provides a way to communicate with a web server from a web page using JavaScript.
- It can send HTTP requests (GET, POST, PUT, DELETE, etc.) to a server and handle the server's response asynchronously.

**2. Asynchronous Requests:**

- AJAX allows you to make asynchronous requests, meaning that the web page doesn't need to wait for the response before continuing to process other tasks.
- This asynchronous behavior enhances the user experience by preventing the page from freezing during data retrieval.

## 3. Data Formats:

- Originally, AJAX used XML for data interchange, and the acronym itself includes "XML." However, JSON (JavaScript Object Notation) has become the preferred format due to its simplicity, ease of use, and compatibility with JavaScript.
- You can also use other formats like plain text or HTML, depending on your requirements

**4. AJAX Workflow:**

- An AJAX request typically involves the following steps:
  1. Create an XMLHttpRequest object.
  2. Define a callback function to handle the server's response.
  3. Open a connection to the server, specifying the HTTP method and URL.
  4. Send the request to the server.
  5. Receive and process the server's response in the callback function.

## 5. Callback Functions:

- Callback functions are essential in AJAX to handle server responses.
- Common callback functions include `onreadystatechange`, which monitors the state of the request, and functions like `onload`, `onerror`, and `ontimeout`, which handle specific aspects of the response.

**6. Same-Origin Policy:**

- AJAX requests are subject to the same-origin policy, which means that web pages can only make requests to the same domain from which they originated. This policy is in place for security reasons.
- To make requests to different domains, techniques like JSONP (JSON with Padding) or Cross-Origin Resource Sharing (CORS) are used.

**7. Libraries and Frameworks:**

- While you can work with AJAX directly using the XMLHttpRequest object, many libraries and frameworks simplify AJAX interactions. jQuery's AJAX functions, Axios, and the Fetch API are examples of tools that make it easier to work with AJAX.

**8. Use Cases:**

- AJAX is commonly used for various purposes, including:
- Loading dynamic content without refreshing the entire page (e.g., single-page applications).
- Implementing auto-suggest search boxes.
- Submitting form data in the background to provide instant feedback.
- Fetching data from APIs to populate web pages with real-time information.
- Implementing chat applications and social media features.

## 9. Benefits:

- AJAX enhances user experience by making web pages more interactive and responsive.
- It reduces server load by fetching only the necessary data, rather than reloading entire pages.
- It allows for the creation of web applications that behave more like desktop applications.

**10. Considerations:**

- When implementing AJAX, consider accessibility, error handling, and user feedback to ensure a robust and user-friendly experience.
- Always validate and sanitize data received from the server to prevent security vulnerabilities like cross-site scripting (XSS).

**Note:** summary, AJAX is a powerful technology that enables asynchronous communication between web pages and servers, enhancing the interactivity and responsiveness of web applications. It has become an essential tool in modern web development, allowing developers to create dynamic and user-friendly web experiences.

# JS JSON

- JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. JSON is often used to transmit data between a server and a web application or between different parts of an application. It is a text-based format that resembles JavaScript object literal syntax.

❖ **JSON Syntax:**

- JSON consists of key-value pairs, where keys are strings enclosed in double quotes, followed by a colon, and then values. Values can be strings, numbers, objects, arrays, `true`, `false`, or `null`.
- JSON objects are enclosed in curly braces `{}`, and JSON arrays are enclosed in square brackets `[]`

**Example JSON Data:**

```
{
 "name": "sangam kumar",
 "age": 30,
 "city": "patan",
 "isStudent": false,
 "hobbies": ["reading", "swimming"]
}
```

## ❖ JavaScript and JSON:

- JavaScript provides methods for working with JSON data:
- `JSON.stringify(obj)`: Converts a JavaScript object into a JSON string.
- `JSON.parse(json)`: Parses a JSON string and returns a JavaScript object.

❖ **JSON.stringify() Example:**

```
const person = {
 name: "John Doe",
 age: 30,
 city: "New York",
 isStudent: false,
 hobbies: ["reading", "swimming"]
};
const jsonPerson = JSON.stringify(person);
console.log(jsonPerson);
```

❖ **JSON.parse() Example:**

```
const jsonPerson =
'{"name":"Sangam","age":30,"city":"patan","isStudent":false,"hobbies":["reading","swimmin
g"]}';
const person = JSON.parse(jsonPerson);
console.log(person);
```

## Complete Example:

➢ **Raj.html**

```
<!DOCTYPE html>
<html>
<head>
  <title>JSON Example</title>
</head>
<body>
```

```
    <h1>JSON Example</h1>
    <p id="json-output"></p>
    <script>
        // JavaScript object
        const person = {
            name: "Sangam kumar",
            age: 30,
            city: "patna",
            isStudent: false,
            hobbies: ["reading", "swimming"]
        };
        // Convert JavaScript object to JSON string
        const jsonPerson = JSON.stringify(person);
        document.getElementById('json-output').innerText = 'JSON Data:\n' + jsonPerson;
        // Parse JSON string back to JavaScript object
        const parsedPerson = JSON.parse(jsonPerson);
        console.log(parsedPerson);
    </script>
</body>
</html>
```

# JS vs jQuery

- JavaScript (JS) and jQuery are both tools used for web development, but they serve different purposes and have their own strengths and weaknesses. Let's compare JavaScript and jQuery:

## ❖ JavaScript:

**1. Core Web Language:**

- JavaScript is a fundamental web programming language. It is supported by all modern web browsers, and it's the foundation for building interactive web applications.

**2. DOM Manipulation:**

- JavaScript allows you to directly interact with the Document Object Model (DOM), which represents the structure of a web page. You can access and manipulate HTML elements, attributes, and styles with pure JavaScript.

**3. Full Control:**

- JavaScript provides complete control over every aspect of a web page, making it highly versatile for complex web applications

**4. Community and Ecosystem:**

- JavaScript has a vast and active community, along with a wide range of libraries and frameworks (e.g., React, Angular, Vue.js) that can be used to simplify and enhance web development.

**5. Learning Curve:**

- JavaScript can have a steeper learning curve, especially for beginners, due to its complex features and the need to manage browser compatibility.

**6. Performance:**

- When used efficiently, JavaScript can offer excellent performance as it allows you to optimize code for specific use cases.

## ❖ jQuery:

**1. JavaScript Library:**

- jQuery is a popular JavaScript library that simplifies many common tasks in web development. It is built on top of JavaScript and provides a higher-level API for working with the DOM.

**2. DOM Manipulation Simplified:**

- jQuery abstracts the complexities of DOM manipulation, allowing developers to perform common operations with less code. For example, selecting and manipulating elements is more concise.

**3. Cross-Browser Compatibility:**

- jQuery handles many cross-browser compatibility issues under the hood, making it easier to write code that works consistently across different browsers.

**4. Extensive Plugin Ecosystem:**

- jQuery has a vast ecosystem of plugins that extend its functionality, making it easy to add features like sliders, carousels, and UI components to websites.

**5. Rapid Development:**

- For quick prototypes or smaller projects, jQuery can significantly speed up development, reducing the amount of code that needs to be written.

**6. Learning Curve:** jQuery has a lower learning curve compared to pure JavaScript, making it accessible to beginners and those who want to quickly enhance their websites.

# JS Graphics

- JavaScript Graphics allows you to create and manipulate graphical elements on web pages, enabling the development of interactive and visually appealing web applications. Graphics in JavaScript are typically created using the HTML5 <canvas> element or SVG (Scalable Vector Graphics). Below, I'll provide an overview of both approaches, along with examples using HTML and CSS:

**1. HTML5 <canvas> for Raster Graphics:**

- The <canvas> element provides a raster graphics environment where you can draw and manipulate pixel-based images. Here are some key points:
- Drawing Context: To work with <canvas>, you obtain a drawing context using getContext('2d'). The 2D context (ctx) provides methods and properties for drawing shapes, text, and images.
- Shapes: You can draw various shapes, including rectangles, circles, lines, and paths, on the canvas. These shapes can be filled with colors or patterns.
- Images: You can load and display images on the canvas using the drawImage method, allowing you to create image galleries, games, and more.
- Text: The canvas allows you to render text with different fonts, sizes, and styles. You can position text wherever you like on the canvas.
- Animation: JavaScript can be used to create animations on <canvas>. By repeatedly redrawing the canvas at specific intervals, you can create interactive animations and games.
- Event Handling: You can handle user interactions such as mouse clicks, mouse movement, and keyboard input to create interactive graphics.

**Example:**

```
<!DOCTYPE html>
<html>
<head>
  <title>Canvas Graphics Example</title>
  <style>
    canvas {
      border: 1px solid #000;
    }
  </style>
</head>
<body>
  <h1>Canvas Graphics Example</h1>
  <canvas id="myCanvas" width="400" height="200"></canvas>
  <script> // Get the canvas element and its drawing context
    const canvas = document.getElementById('myCanvas');
    const ctx = canvas.getContext('2d');// Draw a red rectangle
    ctx.fillStyle = 'red';
    ctx.fillRect(50, 50, 200, 100);// Draw text
    ctx.fillStyle = 'blue';
    ctx.font = '24px Arial';
    ctx.fillText('Hello, Canvas!', 100, 180);
  </script>
</body></html>
```

## 2. SVG (Scalable Vector Graphics) for Vector Graphics:

- SVG is a markup language that describes vector graphics using XML. It offers a powerful way to create graphics that can be scaled without loss of quality. Here are some key points:
- Vector Graphics: SVG is ideal for creating vector graphics, which are based on mathematical formulas that describe shapes. This means that SVG images can be scaled up or down without losing detail or quality.
- Elements: SVG uses XML elements to define graphics. Common SVG elements include <rect> for rectangles, <circle> for circles, <path> for custom shapes, <text> for text, and more.
- Attributes: SVG elements are styled using attributes like fill (for colors), stroke (for outlines), stroke-width (for line thickness), and others.
- Interactive: SVG graphics can be made interactive by adding event listeners to SVG elements, allowing you to respond to user actions like clicks and mouseovers.
- Animation: SVG supports animations and transitions using CSS, JavaScript, or declarative animations defined within the SVG itself.
- Accessibility: SVG images are accessible by default, making them a good choice for creating graphics that are compatible with screen readers and assistive technologies.
- Integration: SVG graphics can be embedded directly into HTML documents as inline SVG or included as external files. This makes SVG a flexible choice for creating illustrations, icons, charts, and more.

### Example:

```
<!DOCTYPE html>
<html>
<head>
  <title>SVG Graphics Example</title>
</head>
<body>
  <h1>SVG Graphics Example</h1>
  <svg width="400" height="200">
    <rect x="50" y="50" width="200" height="100" fill="red" />
    <text x="100" y="180" font-family="Arial" font-size="24" fill="blue">Hello, SVG!</text>
  </svg>
</body>
</html>
```

# Exception Handling in js:

- Exception handling in JavaScript is a mechanism that allows developers to gracefully handle errors and unexpected situations in their code.
- JavaScript provides a set of constructs for handling exceptions to prevent script termination and provide useful information about the error.

**Example:**

## 1. Types of Errors:

- In JavaScript, there are several types of errors, including:
  - **Syntax Errors:** Occur when the code violates the language syntax rules. These errors prevent code execution and are usually caught during script parsing.
  - **Runtime Errors (Exceptions):** Occur during script execution when something goes wrong, such as division by zero, referencing an undefined variable, or trying to access properties of null or undefined objects.
  - **Logical Errors:** Do not generate exceptions but result in incorrect behavior due to flawed logic in code.

### 2. try...catch` Statement:

- The primary mechanism for handling exceptions in JavaScript is the `try...catch` statement.
- It allows you to wrap potentially error-prone code within a `try` block and specify how to handle exceptions in the `catch` block.

**Example:**

```
try {
  // Code that may throw an exception
} catch (error) {
  // Code to handle the exception
}
```

## 3. Error Object:

- When an exception is thrown, JavaScript creates an `Error` object that contains information about the error, including the error message and stack trace.
- You can access this object through the `catch` block's parameter.

**Example:**

```
try {
  // Code that may throw an exception
} catch (error) {
  console.error(error.message);
}
```

### 4. finally` Block:

- The `finally` block, if present, is executed regardless of whether an exception was thrown or caught.
- It is often used for cleanup tasks like closing files or releasing resources.

**Example:**

```
try {
  // Code that may throw an exception
} catch (error) {
  // Code to handle the exception
} finally {// Cleanup code
}
```

**5. Throwing Custom Errors:**

- Developers can throw custom errors using the `throw` statement.
- This allows you to create and propagate custom error messages with specific details.

Example:

```
function divide(a, b) {
  if (b === 0) {
    throw new Error('Division by zero is not allowed.');
  }
  return a / b;
}
```

**6. Built-in Error Types:**

- JavaScript provides several built-in error types, such as `Error`, `SyntaxError`, `TypeError`, `ReferenceError`, and more.
- You can catch specific error types to handle them differently.

Example:

```
try {
  // Code that may throw an exception
} catch (error) {
  if (error instanceof TypeError) {
    console.error('Type error occurred.');
  } else {
    console.error('An error occurred:', error.message);
  }
}
```

**7. Asynchronous Exception Handling:**

- For asynchronous code, such as `setTimeout` or AJAX requests, exceptions do not propagate to the global scope.
- You should handle exceptions within the asynchronous code or use mechanisms like `Promise` rejections or `async/await` to handle errors.

**8. Global Error Handling:**

- You can set a global error handler using `window.onerror` to catch unhandled exceptions and log or display custom error messages.

**Example:**

```
window.onerror = function (message, source, lineno, colno, error) {
  console.error('An unhandled error occurred:', error.message);
  return true; // Prevent default browser error handling
};
```

# Local Storage in js:

- Local Storage is a web storage solution in JavaScript that allows web applications to store key-value pairs in a client's web browser. It provides a simple way to store and retrieve data persistently on the user's device.
- **overview of how to use Local Storage in JavaScript:**

**1. Storing Data:**

- You can store data in Local Storage using the `localStorage.setItem(key, value)` method. Both `key` and `value` should be strings.

**Example:**

```
localStorage.setItem('username', 'raj');
localStorage.setItem('email', 'ra393@.com');
```

**2. Retrieving Data:**

- To retrieve data from Local Storage, use the `localStorage.getItem(key)` method, where `key` is the string associated with the data you want to retrieve.

**Example:**

```
const username = localStorage.getItem('username');
const email = localStorage.getItem('email');
```

**3. Updating Data:**

- You can update existing data by overwriting it with the `setItem` method.
- localStorage.setItem('username', 'Updatedsangam');

**4. Removing Data:**

To remove data, use the `localStorage.removeItem(key)` method.

```
localStorage.removeItem('email');
```

**5. Clearing All Data**:

To remove all data stored in Local Storage, use the `localStorage.clear()` method.

```
localStorage.clear();
```

**6. Checking for Item Existence:**

You can check if a specific item exists in Local Storage using `localStorage.getItem(key)`. If the item is not found, it returns `null`.

```
const hasUsername = localStorage.getItem('username') !== null;
```

**7. Storage Limitations:**

Local Storage has a storage limit of about 5-10 MB, depending on the browser. Attempting to exceed this limit may result in an error.

**8. Data Type Limitation:**

Local Storage stores data as strings. If you need to store complex data types like objects or arrays, you should serialize and deserialize them using `JSON.stringify()` and `JSON.parse()`.

```
const user = {
  name: 'John',
  email: 'john@example.com'
};
localStorage.setItem('user', JSON.stringify(user));
const retrievedUser = JSON.parse(localStorage.getItem('user'));
```

**9. Event Handling**:

Local Storage does not provide built-in event handlers for changes. You can use the `window` object's `storage` event to listen for changes made to the Local Storage from other tabs or windows of the same website.

```
window.addEventListener('storage', function (e) {
  console.log(`Key "${e.key}" was changed from "${e.oldValue}" to "${e.newValue}".`);
});
```

Example:

- **index.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Local Storage Example</title>
  <link rel="stylesheet" href="raj.css">
</head>
<body>
  <div class="container">
    <h1>Local Storage Example</h1>
    <label for="name">Enter your name:</label>
    <input type="text" id="name" placeholder="Your name">
    <button onclick="saveName()">Save Name</button>
    <p id="savedName"></p>
  </div>
  <script src="raj.js"></script>
</body>
</html>
```

- **raj.js**

```
// Function to save the user's name in Local Storage
function saveName() {
  const nameInput = document.getElementById('name');
  const savedName = document.getElementById('savedName');
  // Retrieve the name from the input field
  const name = nameInput.value;
  // Check if the name is not empty
  if (name.trim() !== '') {
    // Save the name in Local Storage
    localStorage.setItem('user_name', name);
    // Update the paragraph to display the saved name
    savedName.textContent = `Your name is: ${name}`;
  } else {
    alert('Please enter a valid name.');
  }
}
// Check if a name is already saved in Local Storage
const storedName = localStorage.getItem('user_name');
if (storedName) {
```

```
    const savedName = document.getElementById('savedName');
    savedName.textContent = `Your name is: ${storedName}`;
  }
```

- **raj.css**

```
body {
  font-family: Arial, sans-serif;
  background-color: #f2f2f2;
  margin: 0;
  padding: 0;
}
.container {
  background-color: #fff;
  max-width: 400px;
  margin: 0 auto;
  padding: 20px;
  border-radius: 5px;
  box-shadow: 0 0 5px rgba(0, 0, 0, 0.2);
}
h1 {
  text-align: center;
}
label {
  display: block;
  font-weight: bold;
}
input[type="text"] {
  width: 100%;
  padding: 10px;
  margin-bottom: 10px;
  border: 1px solid #ccc;
  border-radius: 3px;
}
button {
  background-color: #007bff;
  color: #fff;
  padding: 10px 20px;
  border: none;
  border-radius: 3px;
  cursor: pointer;
}
button:hover {
  background-color: #0056b3;
}
p {
  font-size: 1.2em;
  text-align: center;
  margin-top: 20px;
}
```

# Important question and answer in JavaScript:

1. What is JavaScript?
    - JavaScript is a versatile and widely used programming language for web development.
2. What are the data types in JavaScript?
    - Primitive types: string, number, boolean, undefined, null, symbol (ES6).
    - Reference types: object, array, function.
3. How do you declare variables in JavaScript?
    - Using var, let, or const.
    - What is the difference between let, const, and var?
    - let and const have block scope, while var has function scope.
    - const cannot be reassigned, let and var can.
5. How do you comment in JavaScript?
    - Using // for single-line comments and /* */ for multi-line comments.
6. What is hoisting in JavaScript?
    - Hoisting is the behavior of moving variable and function declarations to the top of their containing scope during compilation.
7. What is a closure?
    - A closure is a function that retains access to variables from its parent scope even after the parent function has finished executing.
8. How do you define a function in JavaScript?
    - Using the function keyword or as arrow functions (ES6).
9. What is callback hell?
    - Callback hell (or Pyramid of Doom) is a situation where multiple nested callbacks make code hard to read and maintain.
10. What is an anonymous function?
    - An anonymous function is a function without a name.
11. What is the this keyword in JavaScript?
    - this refers to the current object in a function or method.
12. Explain event delegation.
    - Event delegation is a technique where a single event handler is used for multiple elements by using event bubbling.
13. How do you create an object in JavaScript?
    - Using object literals, constructor functions, or class syntax (ES6).
14. What is a prototype in JavaScript?
    - A prototype is an object that provides shared properties and methods for other objects.
15. What is the difference between null and undefined?
    - null is an intentional absence of any object value, while undefined means a variable has been declared but has not been assigned a value.
16. How do you check if a variable is undefined?
    - Using typeof or by comparing to undefined.
17. What is the purpose of the NaN value?
    - NaN represents "Not-a-Number" and is a value returned when a mathematical operation cannot produce a valid result.
18. What is the difference between == and ===?
    - == checks for equality after type coercion, while === checks for strict equality (value and type).

19. How do you convert a string to a number in JavaScript?
    - Using parseInt() or parseFloat().
20. What is the NaN value used for?
    - NaN is used to represent the result of an invalid mathematical operation.
21. What is an IIFE?
    - An IIFE (Immediately Invoked Function Expression) is a function that is defined and executed immediately.
22. What is a promise in JavaScript?
    - A promise is an object representing the eventual completion or failure of an asynchronous operation.
23. How do you handle promises in JavaScript?
    - Using .then() for success and .catch() for errors.
24. What is asynchronous programming in JavaScript?
    - Asynchronous programming allows non-blocking execution of code, typically used for tasks like network requests and timers.
25. Explain the event loop in JavaScript.
    - The event loop is a mechanism that handles asynchronous operations, ensuring that the main thread remains responsive.
26. What is the difference between null and undefined?
    - null is explicitly assigned to indicate the absence of a value, while undefined indicates that a variable has been declared but not assigned a value.
27. What is the purpose of the typeof operator?
    - typeof is used to determine the data type of a variable or expression.
28. How do you add an element to an array in JavaScript?
    - Using the push() method or the spread operator (ES6).
29. What is JSON and how do you parse it in JavaScript?
    - JSON (JavaScript Object Notation) is a data interchange format.

    Use JSON.parse() to parse a JSON string into a JavaScript object.
30. What is the ternary operator in JavaScript?
    - The ternary operator (condition ? expr1 : expr2) is a shorthand for an if-else statement.
31. How do you prevent the default behavior of an event in JavaScript?
    - Using event.preventDefault().
32. What is the localStorage and sessionStorage objects used for?
    - They provide storage for web applications on the client side.
33. How do you loop through an array in JavaScript?
    - Using for, while, or forEach().
34. What is a JavaScript constructor function?
    - A constructor function is used to create and initialize objects.
35. How do you define and use classes in JavaScript (ES6)?
    - Use the class keyword to define classes and the new keyword to create instances.
36. What are arrow functions in JavaScript (ES6)?
    - Arrow functions are concise function expressions that do not bind their own this value.
37. What is the purpose of the map() function in JavaScript?
    - The map() function is used to create a new array by applying a function to each element of an existing array.
38. How do you add and remove classes from HTML elements in JavaScript?

- Use .classList.add() and .classList.remove() methods.

39. Explain the difference between the global scope and local scope.
    - Global scope is accessible from anywhere in the code, while local scope is limited to a specific function or block.
40. What is a callback function in JavaScript?
    - A callback function is a function passed as an argument to another function and executed at a later time.
41. What is a higher-order function in JavaScript?
    - A higher-order function is a function that takes one or more functions as arguments or returns a function.
42. How do you clone an object in JavaScript?
    - Using the spread operator ({ ...obj }) or Object.assign().
43. What is destructuring in JavaScript?
    - Destructuring allows you to extract values from arrays or objects into variables.
44. How do you check if an array contains a specific element in JavaScript?
    - Using the includes() method or indexOf().
45. What is the purpose of the fetch() API in JavaScript?
    - fetch() is used for making network requests and retrieving data from web APIs.
46. What is the purpose of the async and await keywords (ES6)?
    - async is used to define asynchronous functions, and await is used to pause execution until a Promise is resolved or rejected.
47. How do you create and use modules in JavaScript (ES6)?
    - Use import and export statements to define and use modules.
48. What is the difference between null and undefined?
    - null is explicitly assigned to indicate the absence of a value, while undefined indicates that a variable has been declared but not assigned a value.
49. What is the purpose of the try...catch statement in JavaScript?
    - try...catch is used for error handling to catch and handle exceptions.
50. How do you use the localStorage and sessionStorage objects to store data on the client side?
    - Use localStorage.setItem() and localStorage.getItem() (and similarly for sessionStorage) to store and retrieve data.
51. How do you remove an item from an array in JavaScript?
    - Using the splice() method or array methods like filter().
52. What is the difference between let and const?
    - let allows reassignment, while const does not.
53. What is the purpose of the reduce() function in JavaScript?
    - reduce() is used to reduce an array to a single value by applying a function to each element.
54. What is the difference between null and undefined?
    - null is explicitly assigned to indicate the absence of a value, while undefined indicates that a variable has been declared but not assigned a value.
55. How do you create and use JavaScript promises?
    - Promises are created using the Promise constructor and used with .then() and .catch().
56. What is the purpose of the event.preventDefault() method?
    - It prevents the default behavior of an event, such as form submission or link navigation.
57. How do you check if an object has a specific property in JavaScript?
    - Using the hasOwnProperty() method or the in operator.
58. What is the purpose of the Set object in JavaScript (ES6)?

- Set is used to store unique values and provides methods for set operations.

59. How do you convert a string to uppercase or lowercase in JavaScript?
    - Using toUpperCase() and toLowerCase() string methods.
60. What is the purpose of the setTimeout() function in JavaScript?
    - setTimeout() is used to schedule a function to run after a specified delay.
61. How do you create a random number in JavaScript?
    - Using Math.random() and mathematical operations.
62. What is the purpose of the Array.isArray() method?
    - It checks if an object is an array.
63. How do you reverse an array in JavaScript?
    - Using the reverse() method.
64. What is the difference between localStorage and sessionStorage?
    - localStorage persists data until explicitly removed, while sessionStorage data is cleared when the session ends.
65. How do you create and use a JavaScript class (ES6)?
    - Use the class keyword to define a class and the new keyword to create instances.
66. What is the purpose of the splice() method in JavaScript?
    - splice() is used to modify an array by adding or removing elements.
67. How do you check if a value is a number in JavaScript?
    - Using typeof or the isNaN() function.
68. What is the purpose of the pop() and push() methods in JavaScript?
    - pop() removes and returns the last element of an array, while push() adds one or more elements to the end.
69. How do you compare two objects for equality in JavaScript?
    - You need to manually compare their properties and values.
70. What is the purpose of the filter() method in JavaScript?

    filter() is used to create a new array with elements that meet a specified condition.

## ❖ Research(अनुसंधान):

- अनुसंधान एक प्रणालीकरण कार्य होता है जिसमें विशेष विषय या विषय की नई ज्ञान एवं समझ को प्राप्त करने के लिए सिद्धांतिक जांच और अध्ययन किया जाता है। इसकी प्रक्रिया में डेटा का संग्रह और विश्लेषण, निष्कर्ष निकालना और विशेष क्षेत्र में मौजूदा ज्ञान में योगदान किया जाता है। अनुसंधान के माध्यम से विज्ञान, प्रोधोगिकी, चिकित्सा, सामाजिक विज्ञान, मानविकी, और अन्य क्षेत्रों में विकास किया जाता है। अनुसंधान की प्रक्रिया में अनुसंधान प्रश्न या कल्पनाएँ तैयार की जाती हैं, एक अनुसंधान योजना डिज़ाइन की जाती है, डेटा का संग्रह किया जाता है, विश्लेषण किया जाता है, निष्कर्ष निकाला जाता है और परिणामों को उचित दर्शाने के लिए समाप्ति तक पहुंचाया जाता है।

## ❖ Innovation(नवीनीकरण): -

- Innovation (इनोवेशन) एक विशेषता या नई विचारधारा की उत्पत्ति या नवीनीकरण है। यह नए और आधुनिक विचारों, तकनीकों, उत्पादों, प्रक्रियाओं, सेवाओं या संगठनात्मक ढंगों का सृजन करने की प्रक्रिया है जिससे समस्याओं का समाधान, प्रतिस्पर्धा में अग्रणी होने, और उपयोगकर्ताओं के अनुकूलता में सुधार किया जा सकता है।

## ❖ Discovery (आविष्कार):

- Discovery का अर्थ होता है "खोज" या "आविष्कार"। यह एक विशेषता है जो किसी नए ज्ञान, अविष्कार, या तत्व की खोज करने की प्रक्रिया को संदर्भित करता है। खोज विज्ञान, इतिहास, भूगोल, तकनीक, या किसी अन्य क्षेत्र में हो सकती है। इस प्रक्रिया में, व्यक्ति या समूह नए और अज्ञात ज्ञान को खोजकर समझने का प्रयास करते हैं और इससे मानव सभ्यता और विज्ञान-तकनीकी के विकास में योगदान देते हैं।

**Note**: अनुसंधान विशेषता या विषय पर नई ज्ञान के प्राप्ति के लिए सिस्टमैटिक अध्ययन है, जबकि आविष्कार नए और अज्ञात ज्ञान की खोज है।

# TWKSAA RID MISSION

| (Research) | (Innovation) | (Discovery) |
|---|---|---|
| **अनुसंधान करने के महत्वपूर्ण कारण:** | **नवीनीकरण करने के महत्वपूर्ण कारण:** | **खोज करने के महत्वपूर्ण कारण:** |
| 1. नई ज्ञान की प्राप्ति | 1. प्रगति के लिए | 1. नए ज्ञान की प्राप्ति |
| 2. समस्याओं का समाधान | 2. परिवर्तन के लिए | 2. ज्ञान के विकास में योगदान |
| 3. तकनीकी और व्यापार में उन्नति | 3. उत्पादन में सुधार | 3. अविष्कारों की खोज |
| 4. विकास को बढ़ावा देना | 4. प्रतिस्पर्धा में अग्रणी होने के लिए | 4. समस्याओं का समाधान |
| 5. सामाजिक प्रगति | 5. समाज को लाभ | 5. समाज के उन्नति का माध्यम |
| 6. देश विज्ञान और प्रौद्योगिकी का विकास | 6. देश विज्ञान और प्रौद्योगिकी के विकास। | 6. देश विज्ञान और तकनीक के विकास |

➢ जो लोग रिसर्च, इनोवेशन और डिस्कवरी करते हैं उन लोगों को ही हमें अपना नायक, प्रतीक एवं आदर्श मानना चाहिए क्योंकि यें लोग हमारे समाज, देश एवं विज्ञान के क्षेत्र में प्रगति, विकास और समस्याओं के समाधान में महत्वपूर्ण भूमिका निभाते हैं।

मैं राजेश प्रसाद एक वीणा उठाया हूँ Research, Innovation and Discovery का जिसका मुख्य उदेश्य हैं आने वाले समय में सबसे पहले New( RID, PMS & TLR) की खोज, प्रकाशन एवं उपयोग भारत की इस पावन धरती से ही हो |

"अगर आप भी Research, Innovation and Discovery के क्षेत्र में रूचि रखतें हैं एवं अपनी प्रतिभा से दुनियां को कुछ नया देना चाहतें तो हमारे इस त्वक्सा रीड मिशन (TWKSAA RID MISSION) से जरुर जुड़ें" |

- **राजेश प्रसाद**