### 1)Console.log() Method

* **Purpose:** Prints output to the console, useful for debugging.
* **Syntax:** console.log(value);
* **Example:**

const name = 'John';

console.log(name); // Output: John

### 2)Variables: let, const

**let**: Block-scoped variable that can be updated but not accessed before declaration.

let x = 10;

if (true) {

let x = 20; // Block-scoped

console.log(x); // 20

}

console.log(x); // 10

**const**: Block-scoped variable that cannot be reassigned and must be initialized when declared.

const pi = 3.14;

// pi = 3.1415; // Error: Cannot reassign

**3)JavaScript Data Types**

1. **String**
   * **Explanation**: Represents a sequence of characters. Can be enclosed in single quotes, double quotes, or backticks (for template literals).
   * **Syntax**: const str = 'Hello, World!';
   * **Example**:

const greeting = "Hello, " + "World!"; // Output: Hello, World!

1. **Number**
   * **Explanation**: Represents both integer and floating-point numbers. JavaScript numbers range from -(2^53 - 1) to 2^53 - 1.
   * **Syntax**: const num = 123;
   * **Example**:

const amount = 25; // Output: 25

1. **BigInt**
   * **Explanation**: Represents integers with arbitrary precision. Denoted by appending an n to the end of the number.
   * **Syntax**: const bigNum = 1234567890123456789012345678901234567890n;
   * **Example**:const bigNumber = 1234567890123456789012345678901234567890n;
2. **Boolean**
   * **Explanation**: Represents a value that is either true or false.
   * **Syntax**: const isTrue = true;
   * **Example**:

const isActive = Boolean(1); // Output: true

1. **Null**
   * **Explanation**: Represents the intentional absence of any value.
   * **Syntax**: const emptyValue = null;
   * **Example**:

const noValue = null;

1. **Undefined**
   * **Explanation**: Represents a variable that has been declared but not assigned a value.
   * **Syntax**: let uninitialized;
   * **Example**:

let value;

console.log(value); // Output: undefined

**For Checking Data Type We USe:**

typeof variableName;

### 4)Explicit Type Conversion

* **To String**: String(value) or value.toString()

let str = String(123); // '123'

* **To Number**: Number(value), parseInt(value), or parseFloat(value)

let num = Number('456'); // 456

* **To Boolean**: Boolean(value)

let bool = Boolean('hello'); // true

### 5)JavaScript Operators

#### **i)Comparison Operators**

* **Greater than:** a > b
* **Less than:** a < b
* **Greater than or equal to:** a >= b
* **Less than or equal to:** a <= b
* **Not equal to:** a != b
* **Equal to:** a == b
* **Strictly equal to:** a === b

**Examples:**

console.log("2" == 2); // true

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

#### **ii)Logical Operators**

* **&& (AND):** operand1 && operand2
* **|| (OR):** operand1 || operand2
* **! (NOT):** !operand

**Examples:**

console.log(true && false); // false

console.log(true || false); // true

console.log(!true); // false

#### **iii)Arithmetic Operators**

* **+ (Addition):** operand1 + operand2
* **- (Subtraction):** operand1 - operand2
* **\* (Multiplication):** operand1 \* operand2
* **/ (Division):** operand1 / operand2
* **% (Modulo):** operand1 % operand2
* **++ (Increment):** operand++
* **-- (Decrement):** operand--

**Examples:**

console.log(5 + 3); // 8

console.log(5 - 3); // 2

console.log(5 \* 3); // 15

console.log(6 / 3); // 2

console.log(5 % 3); // 2

### 6)String Methods and Operations

1. ***Concatenation***
   * **Explanation**: Combines two or more strings into one.
   * **Syntax**: string1 + string2
   * **Returns**: A new string combining the original strings.
   * **Example**:

let a = ' My name is nirajan ';

let b = "nirajan";

console.log(a + b + "Khatiwada"); // Output: ' My name is nirajan nirajanKhatiwada'

1. ***Simple Form (String Boilerplate)***
   * **Explanation**: Uses template literals to embed expressions within a string.
   * **Syntax**: `${expression}`
   * **Returns**: A new string with evaluated expressions.
   * **Example**:

let a = ' My name is nirajan ';

let b = "nirajan";

console.log(`${a}${b}khatiwada`); // Output: ' My name is nirajan nirajankhatiwada'

1. **Accessing Element of String**
   * **Explanation**: Retrieves the character at a specified index.
   * **Syntax**: string[index]
   * **Returns**: The character at the given index (or undefined if out of range).
   * **Example**:

let a = ' My name is nirajan ';

console.log(a[0]); // Output: ' '

1. **Finding Length of String**
   * **Explanation**: Gets the number of characters in the string.
   * **Syntax**: string.length
   * **Returns**: The length of the string as a number.
   * **Example**:

let a = ' My name is nirajan ';

console.log(a.length); // Output: 21

1. **To Uppercase**
   * **Explanation**: Converts all characters in the string to uppercase.
   * **Syntax**: string.toUpperCase()
   * **Returns**: A new string with all characters in uppercase.
   * **Example**:

let a = ' My name is nirajan ';

console.log(a.toUpperCase()); // Output: ' MY NAME IS NIRAJAN '

1. **To Lowercase**
   * **Explanation**: Converts all characters in the string to lowercase.
   * **Syntax**: string.toLowerCase()
   * **Returns**: A new string with all characters in lowercase.
   * **Example**:

let a = ' My name is nirajan ';

console.log(a.toLowerCase()); // Output: ' my name is nirajan '

1. **Finding Index of a Substring**
   * **Explanation**: Finds the first occurrence of a specified substring.
   * **Syntax**: string.indexOf(substring)
   * **Returns**: The index of the first occurrence of the substring (or -1 if not found).
   * **Example**:

let a = ' My name is nirajan ';

console.log(a.indexOf('n')); // Output: 6

1. **String Slicing**
   * **Explanation**: Extracts a section of the string based on start and end indices.
   * **Syntax**: string.slice(start, end)
   * **Returns**: A new string containing the extracted section.
   * **Example**:

let a = ' My name is nirajan ';

console.log(a.slice(0, 4)); // Output: ' My'

1. **Trim**
   * **Explanation**: Removes whitespace from both ends of the string.
   * **Syntax**: string.trim()
   * **Returns**: A new string with whitespace removed from both ends.
   * **Example**:

let a = ' My name is nirajan ';

console.log(a.trim()); // Output: 'My name is nirajan'

1. **Replace**
   * **Explanation**: Replaces the first occurrence of a specified substring or pattern with a new substring.
   * **Syntax**: string.replace(search, replacement)
   * **Returns**: A new string with the specified substring replaced.
   * **Example**:

let a = ' My name is nirajan ';

console.log(a.replace("nirajan", "kirajan")); // Output: ' My name is kirajan '

1. **Split**
   * **Explanation**: Splits the string into an array of substrings based on a separator.
   * **Syntax**: string.split(separator, limit)
   * **Returns**: An array of substrings.
   * **Example**:

let a = ' My name is nirajan ';

console.log(a.split(" ")); // Output: [' My', 'name', 'is', 'nirajan']

1. **Includes**
   * **Explanation**: Checks if a substring is present in the string.
   * **Syntax**: string.includes("substring")
   * **Returns**: true if the substring is found, false otherwise.
   * **Example**:

console.log("hello".includes("he")); // Output: true

### 7)Number Method

1. **toFixed()**
   * **Explanation:** Formats a number using fixed-point notation with a specified number of decimal places.
   * **Syntax:** number.toFixed(digits);
   * **Returns:** A string representing the number with the specified number of decimal places.
   * **Example:**

let c = 10.001;

console.log(c.toFixed(10)); // Output: '10.0010000000'

1. **Math.ceil()**
   * **Explanation:** Rounds a number up to the nearest integer.
   * **Syntax:** Math.ceil(number);
   * **Returns:** The smallest integer greater than or equal to the given number.
   * **Example:**

let a = 1.1000;

console.log(Math.ceil(a)); // Output: 2

1. **Math.floor()**
   * **Explanation:** Rounds a number down to the nearest integer.
   * **Syntax:** Math.floor(number);
   * **Returns:** The largest integer less than or equal to the given number.
   * **Example:**

console.log(Math.floor(a)); // Output: 1

1. **Math.round()**
   * **Explanation: Rounds a number to the nearest integer.**
   * **Syntax: Math.round(number);**
   * **Returns: The value of the number rounded to the nearest integer.**
   * **Example:**

**console.log(Math.round(a)); // Output: 1**

1. **Math.random()**
   * **Explanation: Returns a pseudo-random floating-point number between 0 (inclusive) and 1 (exclusive).**
   * **Syntax: Math.random();**
   * **Returns: A floating-point number between 0 (inclusive) and 1 (exclusive).**
   * **Example:**

**console.log(Math.random()); // Output: A random number between 0 and 1**

### 8)Non-Primitive Data Types in JavaScript

1. **Object**

**Explanation:** Objects are collections of key-value pairs. Keys are usually strings (or symbols) and values can be any data type.

**Syntax:**

let objectName = {

key1: value1,

key2: value2,

// more key-value pairs

};

**Example:**

let data = {

"name": "nirajan",

"age": 20

};

1. **Array**

**Explanation:** Arrays are ordered collections of values. Values can be of any data type and are accessed by their index.

**Syntax:**

let arrayName = [value1, value2, value3, ...];

**Example:**

let a = ["nirajan", "kirajan", "birajan"];

1. **Function**

**Explanation:** Functions are blocks of code designed to perform a particular task. They can be invoked (called) to execute their code.

**Syntax:**

function functionName(parameters) {

// code to be executed

}

**Example:**

function outer() {

console.log("hi");

}

outer(); // Output: hi

### 9)Array methods

1. **Indexing in Array**
   * **Accessing Elements:**
     + **Description:** Arrays are zero-indexed, so the first element is at index 0.
     + **Returns:** Value of the element at the specified index.
     + **Example:**

console.log(a[0]); // Output: 1

console.log(a[3]); // Output: 4

1. **Slicing in Array**
   * **Slicing:**
     + **Description:** Extracts a section of the array and returns it as a new array.
     + **Syntax:** array.slice(startIndex, endIndex)
     + **Returns:** A new array containing the elements from startIndex up to, but not including, endIndex.
     + **Example:**

console.log(a.slice(0, 2)); // Output: [1, 2]

1. **Length of Array**
   * **Description:** Returns the number of elements in the array.
   * **Returns:** Integer (length of the array).
   * **Example:**

console.log(a.length); // Output: 4

1. **Push**
   * **Description:** Adds one or more elements to the end of the array.
   * **Returns:** The new length of the array.
   * **Example:**

a.push(5);

console.log(a); // Output: [1, 2, 3, 4, 5]

1. **Pop**
   * **Description:** Removes the last element from the array.
   * **Returns:** The removed element.
   * **Example:**

a.pop();

console.log(a); // Output: [1, 2, 3, 4]

1. **Shift**
   * **Description:** Removes the first element from the array.
   * **Returns:** The removed element.
   * **Example:**

a.shift();

console.log(a); // Output: [2, 3, 4]

1. **Unshift**
   * **Description:** Adds one or more elements to the beginning of the array.
   * **Returns:** The new length of the array.
   * **Example:**

a.unshift(0);

console.log(a); // Output: [0, 1, 2, 3, 4]

1. **Join**
   * **Description:** Joins all elements of an array into a string, separated by a specified separator.
   * **Returns:** A string representing the array elements joined by the specified separator.
   * **Example:**

let data = a.join(" ");

console.log(data); // Output: "1 2 3 4"

1. **Concatenation of Two Arrays**
   * **Description:** Merges two or more arrays into one.
   * **Returns:** A new array containing the elements of the original arrays.
   * **Example:**

let a2 = [5, 4, 1, 3, 4];

console.log(a.concat(a2)); // Output: [1, 2, 3, 4, 5, 4, 1, 3, 4]

1. **Sort**
   * **Description:** Sorts the elements of an array in place.
   * **Returns:** The sorted array.
   * **Example:**

a2.sort();

console.log(a2); // Output: [1, 3, 4, 4, 5]

1. **Reverse**
   * **Description:** Reverses the order of the elements in the array.
   * **Returns:** The reversed array.
   * **Example:**

a2.reverse();

console.log(a2); // Output: [5, 4, 4, 3, 1]

1. **Removing Elements from a Specific Position**
   * **Description:** Changes the contents of an array by removing or replacing existing elements.
   * **Syntax:** array.splice(index, numberOfElementsToRemove)
   * **Returns:** An array containing the removed elements.
   * **Example:**

let newData = [1, 2, 3, 4];

newData.splice(1, 2); // Removes 2 elements starting at index 1

console.log(newData); // Output: [1, 4]

1. **Inserting Elements at a Specific Position**
   * **Description:** Inserts elements into the array.
   * **Syntax:** array.splice(index, 0, element1, element2, ...)
   * **Returns:** An array containing the removed elements (empty if no elements were removed).
   * **Example:**

let lasrData = [1, 4];

lasrData.splice(1, 0, 2, 3); // Inserts elements 2 and 3 at index 1

console.log(lasrData); // Output: [1, 2, 3, 4]

1. **Spread Operator (...)**
   * **Description:** Spreads out elements of an array into another array or function arguments.
   * **Returns:** A new array containing the elements spread from the original arrays.
   * **Example:**

let finalData = [...newData, ...lasrData];

console.log(finalData); // Output: [1, 4, 1, 2, 3, 4]

1. **Flat**
   * **Description:** Creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.
   * **Syntax:** array.flat(depth)
   * **Returns:** A new array with the specified depth of nesting flattened.
   * **Example:**

let nestedArray = [1, [2, 3], [4, [5, 6]]];

console.log(nestedArray.flat()); // Output: [1, 2, 3, 4, [5, 6]]

console.log(nestedArray.flat(2)); // Output: [1, 2, 3, 4, 5, 6]

1. **Array Destructuring**
   * **Description:** Allows unpacking values from arrays into distinct variables in a concise and readable way.
   * **Basic Syntax:** const [var1, var2, var3] = array;
   * **Example:**

const numbers = [1, 2, 3];

const [first, second, third] = numbers;

console.log(first); // Output: 1

console.log(second); // Output: 2

console.log(third); // Output: 3

1. **Finding Index of a Substring**
   * **Description:** Finds the first occurrence of a specified substring in a string.
   * **Syntax:** string.indexOf(substring)
   * **Returns:** The index of the first occurrence of the substring (or -1 if not found).
   * **Example:**

let a = ' My name is nirajan ';

console.log(a.indexOf('n')); // Output: 10

1. **in Operator**
   * **Description:** Checks if a property exists in an object.
   * **Syntax:** property in object
   * **Returns:** true if the property exists, otherwise false.
   * **Example:**

let obj = { name: 'Niraj', age: 20 };

console.log('name' in obj); // Output: true

console.log('gender' in obj); // Output: false

### ****10)Object in Js****

### ****i. Defining an Object****

An object in JavaScript is a collection of key-value pairs. Each key (also known as a property) is a unique identifier, and the value can be anything: a string, number, array, function, or even another object.

#### Example:

const myObject = {

name: "Nirajan", // String property

class: "Bachelor", // String property

is\_topper: "No", // Boolean property (as a string)

greet: function (from) { // Method (function inside an object)

console.log(`Welcome ${this.name}. From ${from}`);

}

};

### ****ii. Accessing****

There are two common ways to access properties in an object:

* **Dot Notation**:

console.log(myObject.name); // Output: nirajan

This is the most common and preferred method when you know the property name in advance.

* **Bracket Notation**:

console.log(myObject['name']); // Output: nirajan

Bracket notation is useful when the property name is stored in a variable or when it contains special characters or spaces.

### ****iii)Modifying Object Properties****

You can modify an object's properties using dot or bracket notation:

* **Dot Notation:** Use when you know the property name.

myObject.name = "Kirajan"; // Modifies the 'name' property

* **Bracket Notation:** Use when the property name is dynamic or contains special characters.

myObject['class'] = "Master's"; // Modifies the 'class' property

### ****iv. Adding New Properties****

You can dynamically add new key-value pairs to an object.

myObject.lol = "lol";

console.log(myObject.lol); // Output: lol

### ****v. Using the**** this ****Keyword****

The this keyword inside an object’s method refers to the object itself, allowing you to access its properties.

#### Example:

const person = {

name: "Kirajan",

greet: function() {

console.log(`Hello, my name is ${this.name}`);

}

};

person.greet(); // Output: Hello, my name is Kirajan

### ****vi. Objects Inside Objects****

Objects can contain other objects, allowing you to create complex data structures.

#### Example:

const a = {

details: {

name: "Nirajan",

age: 20

}

};

console.log(a.details.name); // Output: Nirajan

### ****vii.**** Spread Operator (...)

The spread operator lets you copy, merge, or combine objects efficiently.

#### **Copying Properties:**

const original = { name: "Kirajan", class: "Bachelor" };

const copy = { ...original };

console.log(copy); // Output: { name: "Kirajan", class: "Bachelor" }

#### **Merging Objects:**

const info1 = { name: "Kirajan", class: "Bachelor" };

const info2 = { age: 21, is\_topper: true };

const combined = { ...info1, ...info2 };

console.log(combined);

// Output: { name: "Kirajan", class: "Bachelor", age: 21, is\_topper: true }

### ****viii. Object Destructuring****

Destructuring allows you to extract properties from an object and assign them to variables.

#### Example:

const lol = { name: "Nirajan", class: 12, rollno: "11" };

const { name, rollno } = lol;

console.log(name); // Output: Nirajan

console.log(rollno); // Output: 11

Also,Renaming in Destructering

const lol = { name: "Nirajan", class: 12, rollno: "11" }; const { name: studentName, rollno: studentRollNo } = lol; console.log(studentName); // Output: Nirajan console.log(studentRollNo); // Output: 11

### ****8. Object Methods****

Objects can have methods—functions that are properties of the object. These methods can perform actions using the object’s data.

#### Example:

const calculator = {

add: function(a, b) {

return a + b;

},

subtract: function(a, b) {

return a - b;

}

};

console.log(calculator.add(5, 3)); // Output: 8

console.log(calculator.subtract(5, 3)); // Output: 2

### ****11.JavaScript Functions****

 **Function Definitions:**

* **Function Expression:**

const add1 = function(a, b) {

return a + b;

};

*Creates a function and assigns it to a variable. You call the function using the variable name.*

* **Arrow Function:**

const add2 = (a, b) => {

return a + b;

};

*Provides a shorter syntax and does not have its own this context.*

* **Function Declaration:**

function add3(a, b) {

return a + b;

}

*Defines a function with a name. It is hoisted, so it can be called before its declaration.*

*Comparison:* add3 has its own this context, while add1 and add2 do not.

 **Using the Spread Operator:**

function add(...data) {

let sum = 0;

for (let i = 0; i < data.length; i++) {

sum += data[i];

}

return sum;

}

console.log(add(1, 2, 3)); // Output: 6

*The ...data syntax lets the function accept any number of arguments as an array.*

 **Immediately Invoked Function Expression (IIFE):**

(

function add(a, b) {

console.log(a + b);

}

)(2, 3); // Output: 5

*An IIFE is a function that runs immediately after its definition, creating a local scope to avoid affecting the global scope.*

### 12)Control Flow in JavaScript

 **Conditional Statements**:

* **if-else Statement**: Executes code blocks based on a condition.

let a = 2;

if (a === 1) {

console.log(1);

} else if (a === 2) {

console.log(2);

} else {

console.log("None");

}

* **switch Statement**: Evaluates an expression and executes code blocks based on matching case values.

switch (a) {

case 1:

console.log(1);

break;

case 2:

console.log(2);

break;

default:

console.log(3);

}

 **Truthy and Falsy Values**:

* **Falsy Values**: Values that evaluate to false in a boolean context.

console.log(Boolean(false)); // false

console.log(Boolean(0)); // false

console.log(Boolean(-0)); // false

console.log(Boolean(0n)); // false

console.log(Boolean("")); // false

console.log(Boolean(null)); // false

console.log(Boolean(undefined)); // false

console.log(Boolean(NaN)); // false

* **Truthy Values**: Any value that is not falsy.

console.log(Boolean(true)); // true

console.log(Boolean(1)); // true

console.log(Boolean(-1)); // true

console.log(Boolean("hello")); // true

console.log(Boolean(" ")); // true

console.log(Boolean({})); // true

console.log(Boolean([])); // true

console.log(Boolean(function() {})); // true

console.log(Boolean(Symbol())); // true

console.log(Boolean(1n)); // true

 **Nullish Coalescing Operator (??)**: Provides a default value when dealing with null or undefined.

let val1 = null;

let val2 = val1 ?? 10;

console.log(val2); // Output: 10

 **Ternary Operator (?:)**: A shorthand for the if-else statement.

let c = 10;

let b = 10;

let largest = (c > b) ? c : b;

console.log(largest); // Output: 10

### 13)Loops in JavaScript

 **For Loop**

// Syntax: for(initialization; condition; increment/decrement) { ... }

for (let i = 0; i < 10; i++) {

console.log(i);

}

 **While Loop**

// Syntax: while(condition) { ... }

let i = 0;

while (i < 10) {

console.log(i);

i++;

}

 **Do-While Loop**

// Syntax: do { ... } while(condition);

let i = 0;

do {

console.log(i);

i++;

} while (i < 10);

 **For-Of Loop**

* **Usage**: Iterates over arrays and strings

const array = [1, 2, 3];

for (const x of array) {

console.log(x);

}

 **For-In Loop**

* **Usage**: Iterates over the properties of an object, indices of an array, or characters of a string

const obj = {

"name": "nirajan",

"lol": "lol"

};

const arr = ["nirajan", "lol"];

const str = "mynameisnirajan";

for (const key in obj) {

console.log(key); // Prints the keys of the object

}

for (const index in arr) {

console.log(arr[index]); // Prints the values of the array

}

for (const index in str) {

console.log(str[index]); // Prints the characters of the string

}

 **For-Each Loop**

* **Usage**: Iterates over array elements

const array = [1, 2, 3];

array.forEach((data) => {

console.log(data);

});

 **Break and Continue Statements**

* **break**: Exits the loop
* **continue**: Skips the current iteration and continues with the next iteration

### 14.Map, Filter, and Reduce in JavaScript

i. **filter()**

* **Purpose**: Creates a new array with elements that pass a test.
* **Example**: To get all odd numbers from an array:

const a = [1, 2, 3, 4, 5, 6];

const filtered = a.filter(num => num % 2); // [1, 3, 5]

### ii. map() Method

* **Purpose**: map() creates a new array populated with the results of calling a provided function on every element in the calling array. It’s used to transform each element in the array.
* **Example**: Create an array of squares from an existing array.

const a = [1, 2, 3, 4, 5, 6];

const squares = a.map((num) => num \* num);

console.log(squares); // Output: [1, 4, 9, 16, 25, 36]

Here, the function num \* num is applied to each element, resulting in a new array of squared numbers.

### iii. reduce() Method

* **Purpose**: reduce() executes a reducer function on each element of the array, resulting in a single output value. It’s used to accumulate or combine values from the array into a single result.
* **Example**: Sum up all the numbers in the array.

const a = [1, 2, 3, 4, 5, 6];

const sum = a.reduce((accumulator, currentValue) => accumulator + currentValue, 0);

console.log(sum); // Output: 21

Here, accumulator starts at 0 and currentValue iterates over each element, summing them up.

### iv. Method Chaining

* **Purpose**: You can chain filter(), map(), and other array methods together to perform multiple operations in a single, readable statement.
* **Example**: Filter out odd numbers and then square them.

const a = [1, 2, 3, 4, 5, 6];

const result = a.filter((num) => num % 2)

.map((num) => num \* num);

console.log(result); // Output: [1, 9, 25]

Here, filter() first selects the odd numbers, and then map() squares those numbers, producing a new array with the squared values of the odd numbers.

### Summary

* **filter()**: Selects elements that meet a specific condition.
* **map()**: Transforms elements based on a function.
* **reduce()**: Reduces the array to a single value based on a function.
* **Method Chaining**: Combines multiple array operations in a concise and readable manner.

These methods are powerful tools for processing and transforming arrays in JavaScript.

### 15.Importing and Exporting in JavaScript

#### **Default Export**

* **Only one default export is allowed per module.**
* Use for the primary function, class, or object in a module.

// utils.js

export default function primaryFunction() {

console.log("This is the primary function");

}

// main.js

import primaryFunction from './utils.js';

primaryFunction(); // Output: This is the primary function

#### **Named Export**

* **Allows multiple exports per module.**
* Use to export multiple functions, variables, or objects.

// utils.js

function function1() { console.log("This is function1"); }

function function2() { console.log("This is function2"); }

export { function1, function2 };

// main.js

import { function1, function2 } from './utils.js';

function1(); // Output: This is function1

function2(); // Output: This is function2

### Summary

* **Default Export**: One per module, no curly braces during import.
* **Named Export**: Multiple per module, use curly braces during import.

### 16.Error Handling

**Try-Catch**

The try...catch statement is used for error handling in JavaScript. It allows you to catch exceptions and handle them gracefully without breaking the execution of the program.

try {

// Code that may throw an error

let result = riskyOperation();

} catch (error) {

// Code to handle the error

console.error('An error occurred:', error);

}

**Throwing Errors in JavaScript:**

* **Purpose:** Use the throw statement to create custom error messages and stop code execution.
* **Syntax:** throw new Error('Error message');

### 17.Timers

**i)setTimeout**

Schedules a function to be executed after a specified delay (in milliseconds).

const timeoutId = setTimeout(() => {

console.log('Executed after 1 second');

}, 1000);

**ii)setInterval**

Repeatedly executes a function at specified intervals (in milliseconds).

const intervalId = setInterval((a, b) => {

console.log(a); // Output: hi

console.log(b); // Output: oi

}, 20, "hi", "oi");

**iii)clearInterval**

Stops a function from being executed repeatedly by clearing the interval.

clearInterval(intervalId);

**iv)clearTimeout**

Cancels a timeout previously established by setTimeout.

clearTimeout(timeoutId);

### 18.Promises

**Promises** represent the eventual completion (or failure) of an asynchronous operation and its resulting value. They are used to handle asynchronous operations in JavaScript.

**i)Creating a Promise**

const myPromise = new Promise((resolve, reject) => {

if (/\* some condition \*/) {

resolve('Success!');

} else {

reject('Failure!');

}

});

**ii)Using Promises**

myPromise.then(result => {

console.log(result); // Success!

}).catch(error => {

console.error(error); // Failure!

});

### 19.Async/Await

**Async/Await** is syntactic sugar over Promises, making asynchronous code easier to write and read.

Note:use Async function if it has await inside it and also we can await promises only

async and await are used in JavaScript to handle asynchronous operations more easily:

* **async**: Declares a function that returns a Promise.
* **await**: Pauses the execution of an async function until the Promise resolves and returns the result.

**Example**:

async function fetchData() {

let data = await someAsyncOperation(); // Waits for the promise to resolve

console.log(data);

}

This makes asynchronous code easier to read and write compared to using Promises directly.

**Using Async/Await**

async function fetchData() {

try {

let response = await fetch('https://api.example.com/data');

let data = await response.json();

console.log(data);

} catch (error) {

console.error('Error fetching data:', error);

}

}

### 20.Fetch API

**Fetch** is used to make HTTP requests and returns a promise that resolves to the response of the request.

**Basic Usage**

fetch('https://api.example.com/data')

.then(response => response.json())

.then(data => console.log(data))

.catch(error => console.error('Error:', error));

### 21.Date in JavaScript

In JavaScript, the Date object is used to work with dates and times. Here’s a quick guide on how to use it:

#### **Creating Date Objects**

* **Current Date and Time:**

let myDate = new Date();

console.log(myDate.toString()); // Outputs the full date and time as a string

console.log(myDate.toDateString()); // Outputs the date part only as a string

console.log(myDate.toLocaleString()); // Outputs the date and time in a localized format

* **Custom Date (Year, Month, Day):**

let myCustomDate = new Date(2024, 0, 2); // Month is zero-based (0 = January)

console.log(myCustomDate.toDateString()); // Outputs the custom date

* **Custom Date with Time (Year, Month, Day, Hour, Minute):**

let myCustomDate = new Date(2024, 0, 2, 10, 1);

console.log(myCustomDate.toString()); // Outputs the custom date and time

* **Date from a String:**

let mtCustomDate = new Date("2023-01-12");

console.log(mtCustomDate.toDateString()); // Outputs the date from the string

This is a quick overview of how to create and manipulate dates using JavaScript’s Date object.