JavaScript Notes

1. JavaScript is Dynamically Typed

- JavaScript is a dynamically typed language, meaning you don't need to define the type of a variable.
- The type is automatically assigned based on the value.

Example:

```
let age = 24; // The type is "number" because the value is a number.
age = "Twenty-four"; // Now the type changes to "string".
```

2. Declaring and Updating Variables

Declaration:

You can declare variables using let, const, var, or without any keyword (not recommended).

```
age = 24; // Variable declared without a keyword (global scope).
// var - function scoped
var name = "John";

// let - block scoped
let age = 25;

// const - block scoped, immutable
const PI = 3.14159;
```

Updating Variables:

Variables can be updated anytime.

```
age = age + 1; // Increment the value of age by 1.
console.log(age); // Output: 25
```

3. Common Data Types

Primitive Data Types

Primitive types represent single values.

Data Type	Example	Description
Number	24, 3.14	Represents numeric values
String	"Tony Stark"	Represents text data
Boolean	true, false	Represents logical true/false values
Null	null	Intentional absence of any value
Undefined	undefined	Variable declared but not assigned yet
BigInt	123n	Large integers
Symbol	Symbol ("id")	Unique, immutable identifier

Examples:

```
Let age = 14;
console.log(typeof  age);//Output:number

let name = "Paras";
console.log(typeof  name);//Output:string

let bigNum= BigInt(123);//Largeinteger
console.log(bigNum);//Output:123n

let uniqueId= Symbol("Hello");//Unique  identifier
```

```
console.log(uniqueId); // Output: Symbol(Hello)
```

Non-Primitive Data Types

Non-primitive types are objects that can store multiple values.

Object

- Objects store data as key-value pairs.

Eg: 1

```
const student = {
   fullName: "Paras Ramola",
   age: 24,
   cgpa: 8.9,
   isPass: true
};


console.log(student); // Output: { fullName: 'Paras Ramola', age: 24, cgpa: 8.9, isPass: true }
console.log(typeof student); // Output: object’
```

Eg:2

```
let rect={
          length:1,
          breadth:2,
          draw:function(){
                    console.log('Draw rect of length',this.length,this.breadth)
          }
 };
```

Accessing and Updating Object Values:

// Accessing values

```
console.log(student["age"]); // Output: 24
console.log(student.age); // Output: 24
```

```
// Updating values
student["age"] = student["age"] + 1;
console.log(student.age); // Output: 25
```

Dynamic Nature of Objects:

You can add or remove key-value pairs in objects.

Eg:

```
function createRectangle2(len,breadth){
   this.len=len;
   this.breadth=breadth;
   this.draw=function(){
      console.log('Draw rect of length',this.len,this.breadth)
   }
}
let obj2= new createRectangle2(3,7);
//Adding
obj2.color="pink";
console.log(obj2);//createRectangle2 {len: 3, breadth: 7, color: 'pink', draw: ƒ}
//Deleting
delete obj2.color;
console.log(obj2); //createRectangle2 {len: 3, breadth: 7, draw: ƒ}
```

//Constructor:

Tells the constructor of a function.

Eg:

(i)

```
obj2.constructor;
//Output:ƒ createRectangle2(len,breadth){
    this.len=len;
    this.breadth=breadth;
    this.draw=function(){
        console.log('Draw rect of length',this.len,this.breadth)
    }
}
```

```
(ii) createRectangle2.constructor;
//output:
ƒ Function() { [native code] }  // inbuilt constructor of a function is ‘Function’
```

NOTE: functions and arrays are also objects.

Finding a key in objects:

Eg: let car={ color:’pink’ };

```
if( 'color'  in car){
    console.log(“present”,car[ “color” ]);
}else{
    console.log("absent");
}
```

Difference between Primitive and Object type:

Primitive type are copied by their value.

Eg

```
let a=10;
let b=a;
a++;
```

```
console.log(a); //     11

console.log(b); //     10
```

Object are copied by their addresses.

Eg:

```
let temp={value:10};

let a=temp;

temp.value++;

console.log(temp.value); // 11

console.log(a.value);// 11
```

4. Variables with let and const

let

- Block-scoped: Only accessible inside the block {} where it is declared.
- Can be updated but **cannot be redeclared** in the same scope.

Example:

```
let a = 24;
a = 34; // Allowed: Updating the value
console.log(a); // Output: 34
```

var:

var allows redeclaration
Allows updation

Eg:

```
var a=90
console.log(a) //90
Var a=‘hello’
```

```
console.log(a) //hello
```

const

- Block-scoped.
- **Cannot be updated or redeclared.**
- Use const for values that shouldn't change.

Example:

```
const PI = 3.14;
// PI = 90; // Error: Cannot reassign a constant
console.log(PI); // Output: 3.14
```

NOTE:

If you declare a variable without using let, const, or var, it becomes an **implicitly global variable** in non-strict mode. This is generally considered bad practice because it can lead to unexpected behavior and make debugging difficult.

Example:

```
function test() {
  x = 10;  //  No let, const, or var
  console.log(x); // Outputs: 10
}

test();
console.log(x); // Outputs: 10 (x becomes a global variable)
```

Issues with this approach:

1. **Global namespace pollution**:

   The variable x is added to the global window object (in browsers) or global object (in Node.js), increasing the risk of accidental overwrites.

5. Scope of Variables

Block Scope (let and const):

Variables declared inside a block {} are only accessible inside that block.

```
{
  const apple = "red";
  console.log(apple); // Output: red
}
// console.log(apple); // Error: apple is not defined
```

Global Scope (without let or const):

Variables declared without a keyword are accessible everywhere. Avoid using this practice as it can lead to unexpected behaviour.

```
age = 24; // Global variable
console.log(age); // Accessible everywhere
```

6. Summary of Important Points

1. **Primitive Data Types**: Number, String, Boolean, Null, Undefined, BigInt, Symbol.
2. **Non-Primitive Data Types**: Objects (e.g., Arrays, Functions).
3. Use **let** for variables that can change and **const** for constants. Avoid using var.
4. Always initialize variables to avoid undefined.

5. **Objects in const**: While you cannot reassign an object declared with const, you **can update its properties**.

NOTE: ** **console.log()** in JavaScript **automatically moves to the next line** after printing its output. This is because console.log() adds a newline character (\n) after the output.**

Objects Cloning:

let obj1 = { name: "Alice", age: 25 };
let obj2 = obj1; ,
does not create a copy of the object. Instead, it creates a **reference** to the same object in memory.

Shallow Copy:

A **shallow copy** creates a new object, but **nested objects (objects inside object) are still linked** to the original object.
This means **changes in nested objects affect both the original and the copied object**. (only for object not for Primitve type they are independent of original object).

(i) Cloning Using Iteration

Manually copying properties using a loop.

Example:

```
let original = { name: "Alice", age: 25 };

let cloned = {};
for (let key in original) {
   cloned[key] = original[key];
}
```

```
console.log(cloned); // { name: 'Alice', age: 25 }

original.name=“paras”;
console.log(cloned); // { name: 'Alice', age: 25 }
```

(ii) Using Object.assign()

Object.assign(target, source) copies properties from source to target.

Example:

```
let original = { name: "Bob", age: 30 };
let cloned = Object.assign({}, original);

console.log(cloned); // { name: 'Bob', age: 30 }
```

(iii)Cloning Using the Spread Operator (...)

The spread operator { ...original } spreads the properties into a new object.

Example:

```
let original = { name: "Charlie", age: 35 };
let cloned = { ...original };

console.log(cloned); // { name: 'Charlie', age: 35 }
```

Deep Copy

A **deep copy** creates a completely independent clone, including nested objects. Changes in the copied object **do not** affect the original object.

Garbage Collection in JavaScript

Garbage collection (GC) in JavaScript is the process of automatically reclaiming memory by **removing objects that are no longer reachable**. JavaScript uses a built-in **Garbage Collector** to manage memory efficiently, so developers don't need to manually free up memory (unlike C or C++).

Common Math Functions

```
Math.random()  ;              // generate random number
Math.abs(-5);       // 5
Math.max(1, 3, 2);  // 3
Math.min(1, 3, 2);  // 1
Math.sin(Math.PI/2); // 1
Math.log(10);       // Natural log
Math.log10(100);    // 2 (log base 10)
```

Operators

Operators in JavaScript are used to perform operations on variables and values.

1. Arithmetic Operators

These operators perform basic mathematical operations.

Operator	Description	Example	Output
+	Addition	a + b	17
-	Subtraction	a - b	1
*	Multiplication	a * b	72
/	Division	a / b	1.125
%	Modulus (remainder)	a % b	1

**	Exponentiation	a ** b	43046721

Example Code:

```
Let a = 9;
let b = 8;
console.log("a + b =", a + b); // Output: 17
console.log("a ** b =", a ** b); // Output: 43046721
```

2. Unary Operators

Used to increment or decrement the value of a variable.

Operator	Description	Example	Output
a++	Post-Increment	console.log(a++)	Prints a, then increments
++a	Pre-Increment	console.log(++a)	Increments, then prints a
a--	Post-Decrement	console.log(a--)	Prints a, then decrements
--a	Pre-Decrement	console.log(--a)	Decrements, then prints a

Example Code:

```
let a = 9;
console.log("a++ =", a++); // Output: 9, then a becomes 10
console.log("++a =", ++a); // Output: 11
```

3. Assignment Operators

These operators assign values to variables, often with additional operations.

Operator	Description	Example	Equivalent
=	Assign	a = 5	a = 5
+=	Add and assign	a += 4	a = a + 4
-=	Subtract and assign	a -= 2	a = a - 2
*=	Multiply and assign	a *= 3	a = a * 3
/=	Divide and assign	a /= 2	a = a / 2
**=	Exponentiation and assign	a **= 2	a = a ** 2

Example Code:

```
let a = 9;
a += 4; // a becomes 13
console.log("a =", a);
b **= 2; // b becomes 64
console.log("b =", b);
```

4. Comparison Operators

These operators compare two values and return a boolean (true or false).

Operator	Description	Example	Output
==	Equal to (value only)	5 == '5'	true
!=	Not equal to (value only)	5 != '5'	false
===	Equal to (value and type)	5 === '5'	false
!==	Not equal to (value and type)	5 !== '5'	true

>	Greater than	9 > 8	true
<	Less than	8 < 9	true
>=	Greater than or equal to	9 >= 9	true
<=	Less than or equal to	8 <= 9	true

Special Case: Type Conversion with ==

- When comparing a string containing a number with another number, JavaScript converts the string to a number for ==.
- Use **===** to avoid type conversion.

```
let x = 5;
let y = "5";
console.log(x == y);  // true (type conversion occurs)
console.log(x === y); // false (strict comparison)
```

Special Case: Loose vs. Strict Equality

1. Loose Equality (==):

- The == operator checks if the values are equal.
- If the types are different, JavaScript **converts** one type to another (type coercion).
- For example:

```
let x= 5;
let y= "5";

console.log(x == y); // Output: true (String "5" is converted to        Number 5)
console.log(49 == "49"); // Output: true (String "49" is converted to Number 49)
```

2. Strict Equality (===):

- The === operator checks if the values are equal **and** if their types are the same.
- This avoids unintended type coercion.

- For example:

```
console.log(x === y); // Output: false (x is a Number, y is a String)
console.log(49 == “49”);//Output: false
```

3. Loose vs. Strict Inequality

- Similarly, != checks for inequality with type coercion, while !== checks for inequality **without** type coercion.
- Example:

```
console.log(x != y);  // Output: false (Values are equal after type coercion)
console.log(x !== y); // Output: true  (Different types: Number vs String)
```

Key Takeaway

- Use === and !== to avoid unexpected behaviour caused by type coercion with == or !=.
- Strict operators (===, !==) ensure both **value** and **type** match.

5. Logical Operators

These operators combine multiple conditions.

Operator	Description	Example
&&	Logical AND: True if **both** true	a > b && b < a
||	Logical OR: True if **any** true	a > b || b < a
!	Logical NOT: Inverts true/false	!(a == b)

Example Code:

```
let a = 9, b = 8;
console.log(a > b && b < a); // Output: true
console.log(a == b || b < a); // Output: true
console.log(!(a == b || b < a)); // Output: false
```

6. Ternary Operator

- A shorthand for if-else statements.
- Syntax: condition ? valueIfTrue : valueIfFalse;

Example Code:

```
let age = 45;
let result = age > 18 ? 'adult' : 'not adult';
console.log(result); // Output: adult
```

Explanation:

If age > 18 is true, result becomes 'adult'. Otherwise, it becomes 'not adult'

Conditional Statements

Conditional statements allow you to execute code based on specific conditions.

1. if-else Statement

Example Code:

```
Let mode = 'light';
let color;

if(mode === 'dark'){
        color = 'black';
}else if(mode === 'light')     {
        color = 'white';
} else {
    color = 'default';
}

console.log(color); // Output: white
```

2. switch Statement

Example Code:

```
let fruit = 'apple';


switch (fruit) {
    case 'banana':
        console.log('Fruit is banana');
        break;
    case 'apple':
        console.log('Fruit is apple');
        break;
    case 'mango':
        console.log('Fruit is mango');
        break;
    default:
        console.log('Fruit is not available');
}
// Output: Fruit is apple
```

Loops

Loops are used to iterate over iterable structures like strings, arrays, and objects.

1. For Loop

- A for loop is used when you know the exact number of iterations required.

Example:

```
for (let i = 0; i < 8; i++) {
   console.log("Paras Ramola");
}
// Output: "Paras Ramola" printed 8 times
```

2. While Loop

- A while loop runs as long as the specified condition is true.

Example:

```
let i = 0;
while (i < 100) {
   if (i % 2 === 0) {
        console.log(i); // Prints even numbers from 0 to 98
    }
    i++;
}
```

3. Do-While Loop

- A do-while loop runs **at least once**, regardless of the condition.

Example:

```
let i = 20;
do {
    console.log("Hello");
} while (i < 10);
// Output: "Hello" printed once because the condition is checked after the first iteration
```

4. For-Of Loop

- The for-of loop iterates over the **values** of an iterable object, such as arrays or strings.
- It is **not suitable for objects** as they are not iterable.

Syntax:

```
for (let  val   of iterable) {
    // Code
}
```

Example: Iterating Over a String

```
let str = "hi world";
let size = 0;

for (let val of str) {
    console.log("val =", val);
    size++;
}

console.log("Size of string:", size);

// Output:
val = h
val =i
val =
val = w
val = o
val = r
val = l
val = d
Size of string=8
```

Note:

- Non-iterable values like numbers will throw an error.

Example:

```
let x = 9089;
for (let val of x) { // Throws an error: x is not iterable
    console.log(val);
}
```

5. For-In Loop

- The for-in loop iterates over the **keys** (or properties) of an object.
- It can also be used for arrays, but it is less common.

Syntax:

```
for (let key in object) {
    // Code
}
```

Example: Iterating Over an Object

```
let student = {
    fname: "Paras Ramola",
    age: 90,
    isPass: true,
    CGPA: 7.92,
};

for (let key in student) {
    console.log("key =", key, ", value =", student[key]);
}
// Output:
// key = fname , value = Paras Ramola
// key = age , value = 90
// key = isPass , value = true
// key = CGPA , value = 7.92
```

Dialog Boxes

- Dialog boxes in JavaScript are pop-up windows used to interact with users by displaying messages, requesting input, or asking for confirmation.

- They temporarily pause script execution until the user responds.

- These include alert, prompt, and confirm.

1. Alert Box

- The **alert()** method displays a simple message in a pop-up dialog box.
- It has only an "OK" button.

- It pauses the execution of the code until the user acknowledges the alert.

Syntax:

```
alert("Message to display");
```

Example:

```
alert("Hey there!");
```

2. Prompt Box

- The prompt() method displays a dialog box with a text input field, allowing the user to enter data.
- ** The input data is always returned as a **string**.
- If the user clicks "Cancel," the method returns null.

Syntax:

```
let input = prompt("Message to display", "Default value (optional)”);
```

Example:

```
let fullname = prompt("Enter your name");
console.log("Your name is:", fullname);
```

3. Confirm Box

A **confirm box** is often used to get the user's approval before performing an action.

- It displays a message with **"OK"** and **"Cancel"** buttons.
- If the user clicks **"OK"**, the confirm() method returns true.

- If the user clicks **"Cancel"**, the method returns false.

Syntax:

```
let Val = confirm("Do you want to continue ?");
```

Example:

```
var Val = confirm("Do you want to continue ?");
if (Val == true) {
        console.log(" CONTINUED!");
        return true;
} else {
        console.log("NOT CONTINUED!");
        return false;
}
```

Question: Check if a Number is Prime (Using Prompt Input) .

```
let isPrime = true;

let n = parseInt(prompt("Enter a number"), 10);

// number is converted to string  by prompt()

//parseInt ->Convert input to an integer


if (n <= 1) {

  console.log(n, "is not a prime number.");

  isPrime = false;

} else if (n === 2 || n === 3) {

  isPrime = true; // 2 and 3 are prime numbers

} else {

  for (let i = 2; i <= Math.sqrt(n); i++) {

    if (n % i === 0) {

      console.log(n, "is not a prime number.");
      isPrime = false;
      break;
    }
  }
}
 if (isPrime) {
  console.log(n, "is a prime number.");
}
```

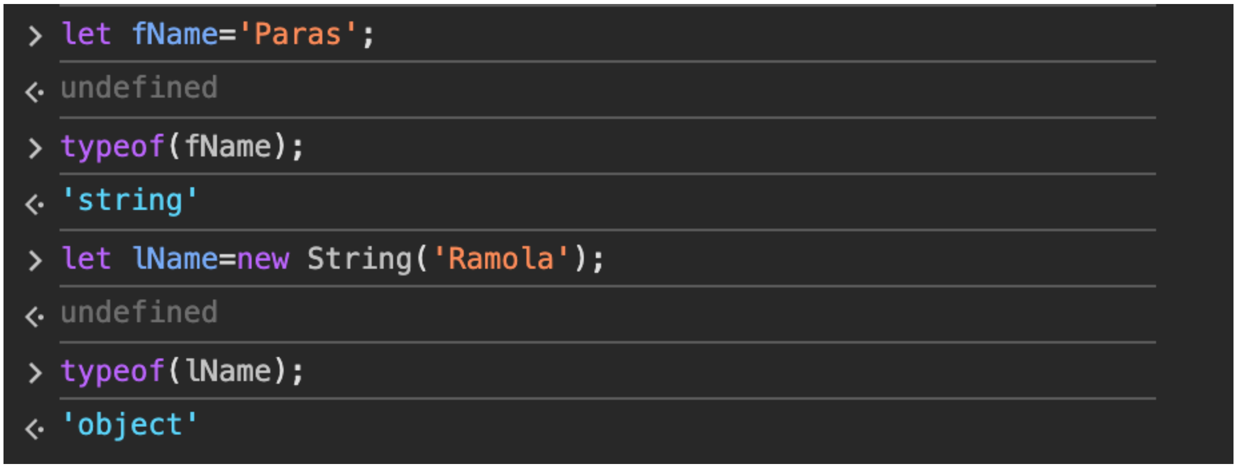
Strings

- A **string** is a sequence of characters used to represent text in JavaScript.
- A string is a primitive data type.
- Strings can be written using:
  - **Double quotes** (" ")
  - **Single quotes** (' ')

Examples

```
let str1 = "paras ramola"; // Double quotes
let str2 = 'paras ramola'; // Single quotes
```

NOTE: A string is a primitive data type but it can be converted to an object type



```
> let fName='Paras';
<· undefined
> typeof(fName);
<· 'string'
> let lName=new String('Ramola');
<· undefined
> typeof(lName);
<· 'object'
```

Basic Operations

1. Length of a String

- Use .length property to find the length of a string.

Example:

console.log(str1.length); // Output: 12

2. String Indices

- Access individual characters using index notation (str[index]).

Example:

console.log(str1[0]); // Output: 'p'

console.log(str1[11]); // Output: 'a'

Template Literals

- Template literals allow embedding expressions and variables directly in strings.
- syntax: ` this is a template literal`

Examples

a) Calculations within Strings

```
let result = `Sum of 2 and 3 is ${2 + 3} `;
console.log(result); // Output: "Sum of 2 and 3 is 5"
console.log(s1.length);// whole ${} is considered one character
```

b) Using Escape Characters

- Example: Newline character (\n)

```
let s = `Hello\nworld`;
console.log(s);
console.log(s2.length);// '\n' is considered as one char
// Output:
// Hello
// world
//11
```

c) String Interpolation

- Substitutes placeholders(${expression}) with variable values.
- Syntax: `string with ${expression}`

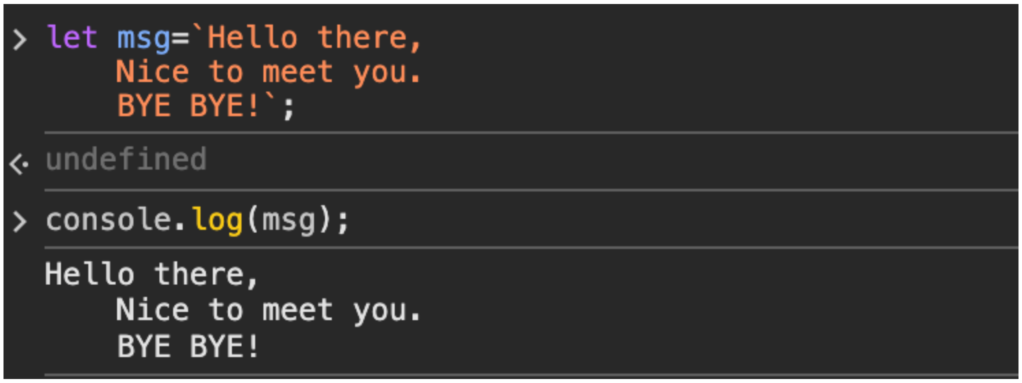
Example:

(i) Refer any variable inside string literal

```
let obj={
item:"Pen",
price:20
};
console.log(`the price of ${obj.item} is ${obj.price}`);
```

// Output: "The price of Pen is 20"

(ii)



```
> let msg=`Hello there,
      Nice to meet you.
      BYE BYE!`;
<· undefined
> console.log(msg);
  Hello there,
      Nice to meet you.
      BYE BYE!
```

String Methods

Immutability of Strings

- Strings are **immutable** in JavaScript.
- Any modification creates a new string; the original string remains unchanged.

1. Convert Case

- str.toUpperCase() - Converts all characters to uppercase.
- str.toLowerCase() - Converts all characters to lowercase.

Example:

```
let s3 = "hello World";
let upperStr = s3.toUpperCase();
console.log(upperStr); // Output: "HELLO WORLD"
console.log(s3);       // Original string remains unchanged
```

2. Remove Whitespace

- str.trim() - Removes whitespace from both the start and end of a string.

Example:

```
let s3 = "   Hello there   ";
console.log(s3.trim()); // Output: "Hello there"
```

3. Slice a String

- str.slice(start, end?) - Extracts a portion of the string.
- end is optional and is **non-inclusive**.

Example:

```
let s3 = "hello world";
console.log(s3.slice(4, 8)); // Output: "o wo"
console.log(s3.slice(5));    // Output: " world"
```

4. Concatenate Strings

- Combine strings using str1.concat(str2) or the + operator.

Example:

```
let s1 = "hello";
let s2 = "world";
console.log(s1.concat(s2)); // Output: "helloworld"
console.log(s1 + " " + s2); // Output: "hello world"
```

5. Replace Characters

- str.replace(searchValue, newValue) - Replaces the first occurrence.
- str.replaceAll(searchValue, newValue) - Replaces all occurrences.

Example:

```
let s1 = "hello";
console.log(s1.replace("l", "y"));     // Output: "heylo"
console.log(s1.replaceAll("l", "y")); // Output: "heyyo"
```

6. Access a Character

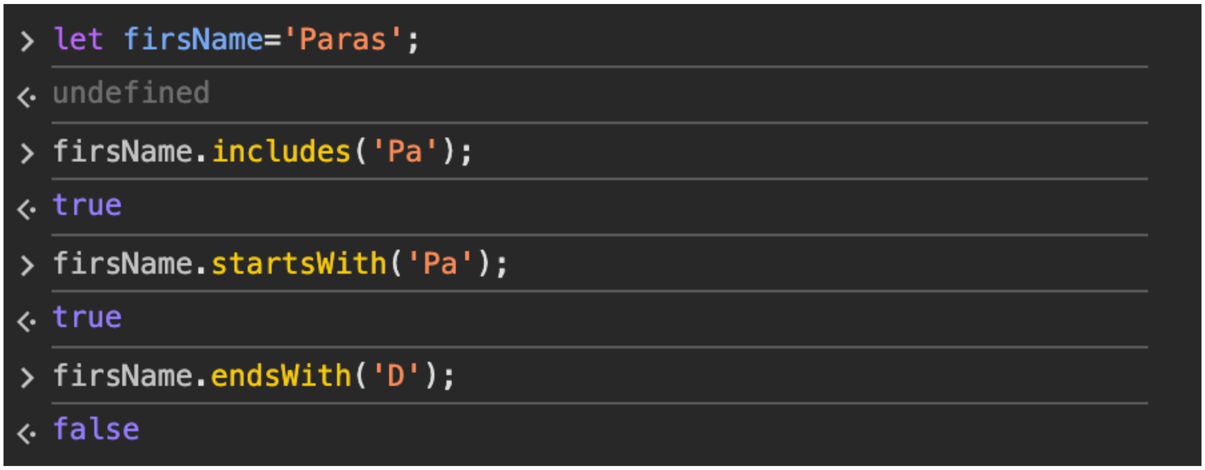
- str.charAt(index) - Returns the character at the specified index.

Example:

```
let s1 = "hello";
console.log(s1.charAt(2)); // Output: “l”
```

6. Spliting:

Other methods:



```
> let firsName='Paras';
<· undefined
> firsName.includes('Pa');
<· true
> firsName.startsWith('Pa');
<· true
> firsName.endsWith('D');
<· false
```

Arrays

- An **array** is a linear collection of items used to store data.
- Arrays are objects in JavaScript, where the keys are indices.
- Arrays can store different types of data in a single array
- Eg: let mixedArray = [42, "hello", true, null, undefined, {name: "John"}, [1, 2, 3]];

Basic Operations

1. Creating an Array

Example:

```
let marks = [97, 68, 56, 89];
console.log(marks);           // Output: [97, 68, 56, 89]
console.log(marks.length);    // Output: 4 (size of the array)
console.log(typeof marks);    // Output: object
```

2. Accessing Values

Example: `console.log(marks[0]); // Output: 97`

3. Mutability of Arrays

- Arrays are **mutable** (unlike strings).

Example:

```
marks[0] = 89; // Update the first element
console.log(marks[0]); // Output: 89
```

Traversing an Array

1. Using for-of

- Loops through the values of the array.

Example:

```
let heroes = ["batman", "superman", "Dad", "spiderman"];
for (let val of heroes) {
   console.log(val);
}
```

```
OUTPUT:
batman
Superman
Dad
spiderman
```

2. Using for-in

- Loops through the indices (keys) of the array.

Example:

```
for (let key in heroes) {
  console.log("key:", key, "value:", heroes[key]);
}
```

OUTPUT:

```
key: 0 value: batman
key: 1 value: superman
key: 2 value: Dad
key: 3 value: spiderman
```

3. Using a for Loop

- Use a traditional for loop for complete control over iteration.

Example:

```
for (let i = 0; i < heroes.length; i++) {
  console.log(heroes[i]);
}
```

OUTPUT:

```
batman
Superman
Dad
spiderman
```

Practice Questions

1. Average Marks

Let marks=[34,67,89,90];

```
let sum = 0;
for (let val of marks) {
    sum += val;
}
let avgMarks = sum / marks.length;
console.log(avgMarks); // Output: Average of marks
```

2. New Prices with a 10% Discount

```
let prices = [50, 645, 300, 900, 50];
let i = 0;
for (let val of prices) {
    let discount = val / 10;
    prices[i] = val - discount;
    console.log(`New price: ${prices[i]}`);
    i++;
}
```

Array Methods

1. push() : Adds elements to the **end** of an array.

Example:

```
let food = ["apple", "pizza"];
food.push("pie");
console.log(food); // Output: ["apple", "pizza", "pie"]

//push multiple item
food.push("burger", "chips");
```

```
console.log(food); // Output: ["apple", "pizza", "pie", "burger", "chips"]
```

2. pop(): Removes the last element and **returns** it.

Example:

```
let lastItem = food.pop();
console.log(lastItem); // Output: "chips"
console.log(food);     // Updated array
```

3. toString(): Converts an array to a comma-separated string.

Example: console.log(food.toString());//Output: "apple,pizza,pie,burger"

4. concat(): **Combines** multiple arrays into a new array.

Example:

```
let arr1 = [45, "hello", 90];
let arr2 = [67, "world", 1];
let combined = arr1.concat(arr2);
console.log(combined); // Output: [45, "hello", 90, 67, "world", 1]

//For multiple  arrays
let       combined2=arr1.concat(arr1,arr2);
console.log(combined2);// [45, "hello", 90, 45, "hello", 90, 67, "world", 1]
```

- First append arr1 to the new array.
- Then, append arr1 again (since you passed it twice).
- Finally, append arr2 to the result.

5. unshift(): Adds elements to the **start** of an array.

Example:

```
arr1.unshift('paras');
console.log(arr1);// Output: [“paras”, 45, "hello", 90];

arr1.unshift("start", 99);//we can add multiple items
console.log(arr1); // Output: ["start", 99,”paras”, 45, "hello", 90]
```

6. shift(): Removes the first element and **returns** it.

Example:

```
let firstItem = arr1.shift();
console.log(firstItem); // Output: "start"
console.log(arr1);      // Updated array
```

7. slice(): Extracts a portion of the array **without modifying** the original array.

syntax: **slice(startIdx,endIdx)**

Example:

```
let sliced = arr1.slice(1, 3); // Start at index 1, end before index 3
console.log(sliced);          // Output: ["hello", 90]

let sliced = arr1.slice(1); // Start at index 1 till end
console.log(sliced);          // Output: ["hello", 90]
```

8. splice()

- Modifies the original array (can add, remove, or replace elements).
- Syntax: splice(startIdx, delCount, newElem1, newElem2, ...)

a) Add and Remove

Example:

```
let ar = [1, 2, 3, 4, 5];
ar.splice(2, 2, 101, 102); // Start at index 2, delete 2 elements  add 101 and 102
console.log(ar);           // Output: [1, 2, 101, 102, 5]
```

b) Only Add

Example:

```
ar.splice(3, 0, 11); // Start at index 3, delete 0 elements, add 11
console.log(ar);     // Output: [1, 2, 101, 11, 102, 5]
```

c) Only Remove

Example:

```
ar.splice(3, 2); // Start at index 3, remove 2 elements,add none
console.log(ar); // Output: [1, 2, 101, 5]
```

c) Replace element

```
 ar.splice(2,1,8);//replace elment at index 2 with 8
 console.log(ar); // Output: [1, 2, 8, 5]
```

**

if we pass only one index->extracts portion of array before index (act as slice)

```
 ar.splice(2);
 console.log(ar); // Output: [1, 2]
```

** ar.splice()//no index->no CHANGE

9. indexOf()

to find the **first index of a value** in an array.

Syntax: array.indexOf(searchElement, fromIndex);

Example:

```
const fruits = ["apple", "banana", "cherry", "banana"];

console.log(fruits.indexOf("banana")); // 1 (first occurrence)
console.log(fruits.indexOf("orange")); // -1 (not found)

console.log(fruits.indexOf("banana", 2)); // 3 (starts search from index 2)
```

If you want the last occurrence, use:

```
fruits.lastIndexOf("banana"); // 3
```

Note: indexOf uses **strict equality (===)**, so:

```
[1, 2, "2"].indexOf(2); // 1
[1, 2, "2"].indexOf("2"); // 2
```

9. includes():

```
const fruits = ["apple", "banana", "cherry", “banana"];

Console.log(fruits.include(“apple”);    //true;
```

9. join():

```
let msg=['we' ,'can' ,’join','elments'];

let joinedMsg=msg.join(‘_');

console.log(joinedMsg);//  we_can_join_elments
```

9.reverse():to reverse an array

Array of Objects:

```
let arr=[
   {name:'rahul',roll:89},
   {name:'Aman',roll:78}
];
console.log(arr.indexOf({name:’rahul',roll:89})); // -1
console.log(arr.includes({name:’rahul',roll:89})); // -1
```

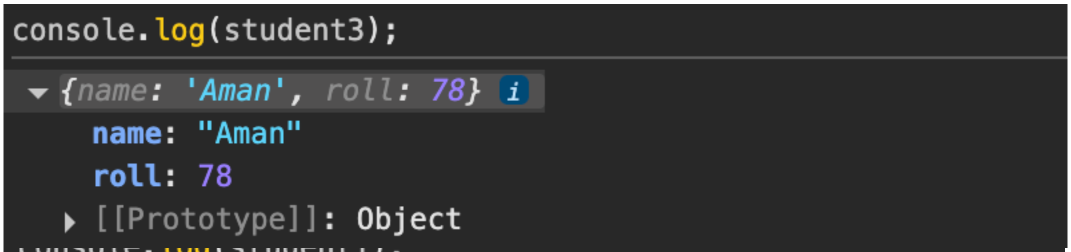
NOTE: indexOf(), includes() doesn’t work with objects because they are not referenced (true adress) while passing them inside above array methods.

Solution: Use Callback functions with **find function**.

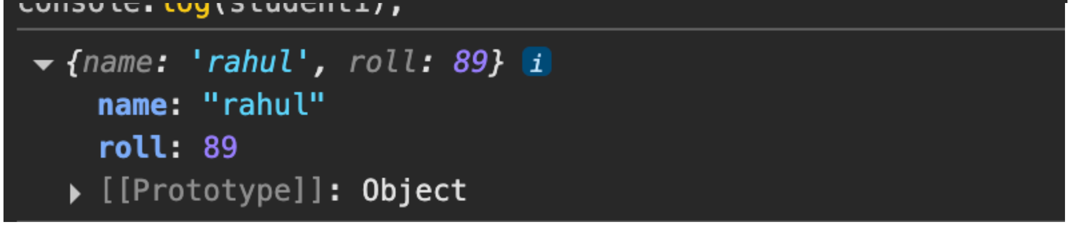
Example:

(i)

```
let student1=arr.find(function(student){
return student.name==='rahul';
});
```

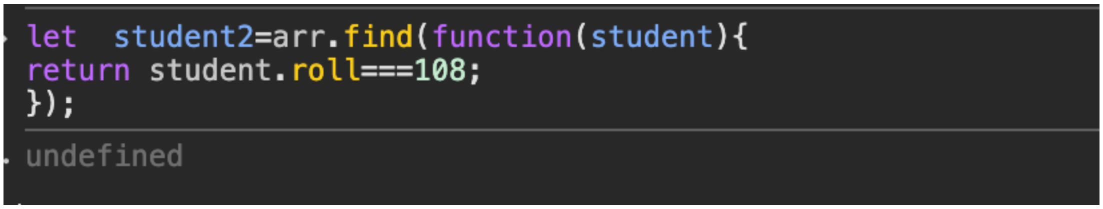


```
console.log(student3);
▾ {name: 'Aman', roll: 78} i
    name: "Aman"
    roll: 78
  ▸ [[Prototype]]: Object
```



```
console.log(student1);
▾ {name: 'rahul', roll: 89} i
    name: "rahul"
    roll: 89
  ▸ [[Prototype]]: Object
```

(ii)



```
let  student2=arr.find(function(student){
return student.roll===108;
});
undefined
```

(iii) using arrow function
let student3=arr.find((student)=>{return student.roll === 78});

(iv) similarly sort for objects

Emptying an Array:

(i) using '[ ]'

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

arr=[];

console.log(arr);

But if :

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

let copy=arr;

```
arr=[];
console.log(arr);// [ ]
console.log(copy);// [1,2,3,4]
```

//copy arr contains the original referenced elements(array is an objects) ,so the elements are not deleted

(ii) using '.lenght=0'

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

(iii) using splice

```
let arr=[1,2,3,4];
let copy=arr;
arr.splice(0,arr.length);
console.log(arr); //  [ ]
console.log(copy); //  [  ]
```

(iv) pop() with loop

Spread Operator:

Used mainly for concat.also used for copying.

Example:

(i)

```
let arr1=[1,2,3,true];

let arr2=['hello',8,9];

let combined1=[ ...arr1 , ...arr2]; // [1, 2, 3, true, 'hello', 8, 9]

let combined2=[ ...arr1 , ...arr2 , true, ’add anything’];  // [1, 2, 3, true, 'hello', 8, 9,’add anything’]
```

(ii) For copying

```
let copy=[...combined2];

console.log(copy2);   // [1, 2, 3, true, 'hello', 8, 9, 'add anyhting']
```

Function

A **function** is a standalone block of code that performs a task or returns a value. **It is not inherently tied to any object.**

Characteristics:

- Defined using ‘ function ‘ keyword or as an arrow function.
- Can be called independently.
- Not associated with any object by default.

Example:

-
  ```
  function greet() {
    console.log("Hello, world!");
  }

  greet(); // Outputs: Hello, world!

  //greet function in not tied to any object
  ```

- console.log(parseInt("123")); // Outputs: 123
  console.log(isNaN("abc"));   // Outputs: true

parseInt() and **isNaN()** are **functions**, as they are not tied to any specific object and   can be called globally.

Method

A **method** is a function that is a property of an object. It is designed to be called in the context of the object it belongs to.

Characteristics:

- Defined as a property of an object.
- Called using the object it belongs to.
- Can access the object's properties using this.

Example:

- **' "abc".toUpperCase() ' , is a method** because '.toUpperCase()' it is tied to the String object.

- Console.log() , log is a method because It is tied to the console object, which provides debugging and logging tools in JavaScript.

Function Definition

Syntax:

```
function funcName(param1, param2) {
   // Do some work
}

funcName();  //function call
```

Note:

- *Parameter*: Variable inside the function definition.
- *Argument*: Value passed during the function call.

Examples:

1. **Without Parameters**:

```
function myfunc() {
console.log("New function");
}
myfunc();
```

2. **With Parameters**:

```
function myfunc1(msg) {
   console.log(msg);
}
myfunc1("I love JS");
```

3. **Function with Return Value**:

```
function sum(a, b) {
   let s = a + b;
   return s;
}

let val = sum(8, 9);
console.log(val); // Outputs: 17

Console.log(a); // ERROR
```

Note: Variables a and b inside the function are local and cannot be accessed outside.

NOTE:

(i) Primitive data types are passed by value in function

Eg:

```
function inc(a){ a++;  }

let temp=10;

inc(temp);

console.log(temp); // 10
```

(ii) Non-Primitive data types (Objects) are passed by reference in function

Eg:

```
function inc(a){

   a.val++;

}

let temp={val:10};

inc(temp);

console.log(temp); // 11
```

Hoisting :

The process of moving function declarations at the top of file automatically.

Eg:The function can be called before declaring-

```
run(); //function call

function run(){

   console.log("run");

}
```

Js will automatically move function declaration at top

Function Assignment:

1.Named Function Assignment:

```
let f = function run(){

            console.log("run");

        }

f( ); // run

Let temp = f;

temp( ); // run
```

2.Anonymous Function Assignment: Function are not named.

```
let f = function (){

            console.log("run");

        }

f( ); // run
```

**Hoisting does not work for function assignment.It only works for function declaration.*

Fixed-Arity Function

```
function sum(a, b){
    console.log(a + b);
}

sum(1, 2);        // Output: 3
sum(1);           // Output: NaN (1 + undefined)
sum(1, 2, 3, 4);  // Output: 3 (only the first two args are used)
```

- JavaScript doesn't enforce the number of arguments.
- If fewer arguments are passed, the missing ones are undefined.
- Extra arguments are ignored unless explicitly handled.

arguments Object

```
function temp(){
   console.log(arguments);
}

temp(1, 2, 3, 4, 5);
```

- Output: *Arguments(5) [1, 2, 3, 4, 5, callee: ƒ, Symbol(Symbol.iterator): ƒ]*

- arguments is an **array-like object**.
- It holds all arguments passed to the function.

Note: arguments is not a real array, so methods like map, filter, etc., don't work directly on it. But you can convert it to an array using Array.from(arguments) or [...arguments].

Example:

```
function sum(){
   let total = 0;
   for(let val of arguments){
      total += val;
   }
   console.log(total);
}

sum(1, 2);        // Output: 3
sum(1);          // Output: 1
sum(1, 2, 3,4);     // Output: 10
```

- This can take any number of arguments.

Rest Operator

The **rest operator** is used in **function parameters** to collect **all remaining arguments** into a single array.

Syntax:

```
function functionName(...restParameter) {
  // restParameter is an array
}
```

Example:

```
function printAll(...args) {

  console.log(args);

}

printAll(1, 2, 3); // Output: [1, 2, 3]
```

NOTE: You can only use **one** rest parameter, and it must be the **last** one.

Example: *rest operator only collects the remaining arguments in a single array.*

```
function greet(first, ...num) {

  console.log(first, num);

}

greet("Hello", "1", "2"); // "Hello"  ["1", “2"]
```

Getters& Setters:

Example:

```
let person = {

  fName: 'Love',

  lName: 'Babbar',
  // Getter for fullName
  get fullName() {
    return `${this.fName} ${this.lName}`;
  },

  // Setter for fullName
  set fullName(value) {
    let parts = value.split(' ');
    this.fName = parts[0];
    this.lName = parts[1];
  }
};
```

Using the Getter:

```
console.log(person.fullName); // Output: "Love Babbar”
```

- Acts like a property
- Automatically runs `get fullName()` when you access `person.fullName`

Using the Setter:

```
person.fullName = "Code Master";
console.log(person.fName); // Output: "Code"
console.log(person.lName); // Output: “Master"
```

- When you **assign** to `fullName`, it automatically runs the `set` method
- It splits the input and assigns to `fName` and `lName`

Notes:

- `get` defines a method that acts like a **property** (no `()` needed)
- `set` lets you define custom logic when **setting** a property

Arrow Functions

A compact way to define functions.

Syntax:

```
const functionName = (param1, param2) => {
   // Do some work
};
```

Examples:

```
const  arrowSum = (a, b) => {
console.log(a+ b);
};
arrowSum(3, 4); // Outputs: 7

const arrowMul = (a, b) => a * b; // Implicit return
console.log(arrowMul(3, 4)); // Outputs: 12
```

Practice Questions:

1. **Count Vowels in a String**: Using a arrow function:

```
const countVowels2=(str)=> {
            let count=0;
            for(let val of str){
            if(val=='a'llval=='e'llval=='i'llval=='o'llval=='u'){
                         count++;
                        }
            }
            return count;
 }
 let cnt1=countVowels2("Hello World");
 console.log(cnt1);
```

Callback Functions

A **callback function** is passed as an argument to another function and executed within it.

1. **Example 1**:

```
function g(){
console.log(“I love JS");
}
function h(callback) {
        return callback;
}

h();        // Error: `h` is missing an argument
h(g);        // Returns the function definition of `g`
h(g)();      // Executes `g` and outputs: "I love JS"
```

2. **Example 2**:

```
function f1() {
   console.log("Hello world");
}

function f2(callback) {
   return callback;
}

f2();       // Error: `f2` is missing an argument
f2(f1);      // Returns the function definition of `f1`
f2(f1)();    // Executes `f1` and outputs: "Hello world"
```

Important Arrays Method

1. forEach

The **forEach()** method iterates over each element of an array and executes a callback function for each element.

Syntax:

array.forEach(callbackFunction(value, index, array));

- **value**: The current element being processed.
- **index** (optional): The index of the current element.
- **array** (optional): The original array.

Examples:

1. **Basic Example**:

```
let arr = [1, 2, 3, 4, 5];

arr.forEach(function (val) {
   console.log(val);
});
// Output: 1, 2, 3, 4, 5
```

2. **Using an Arrow Function**:

```
arr.forEach((val) => {
   console.log(val);
});
```

3. **Using Index and Original Array**:

```
let cities = ["Delhi", "Pune", "Mumbai"];

cities.forEach((val, idx, arr) => {
   console.log(`City: ${val}, Index: ${idx}, Array: ${arr}`);
});
// Output:
// City: Delhi, Index: 0, Array: Delhi,Pune,Mumbai
// City: Pune, Index: 1, Array: Delhi,Pune,Mumbai
// City: Mumbai, Index: 2, Array: Delhi,Pune,Mumbai
```

Interview Question:

What are higher-order functions (HOFs)?

Higher-order functions are functions that:

1. Take other functions as arguments, **or**
2. Return a function as a result.

Example:

```
// forEach is a higher-order function since it takes a callback as an argument.
arr.forEach((val) => console.log(val));
```

2. Map

The map() method creates a new array by applying a transformation to each element in the original array. It does not modify the original array.

Syntax:

array.map(callbackFunction(value, index, array));

Examples:

1. **Creating a New Array**:

```
let nums = [1, 2, 3, 4];

let squaredNums =  nums.map( (val)  => {
        return  val * val
        });
console.log(squaredNums); // Output: [1, 4, 9, 16]
console.log(nums);        // Output: [1, 2, 3, 4] (unchanged)
```

2. **With Index**:

```
let indexedNums =  nums.map( (val, idx)  => val * idx);
console.log(indexedNums); // Output: [0, 2, 6, 12]
```

3. **Mapping with objects**:

```
let num=[1,2,3];

let res=num.map((val)=>{

   return {value:num}

});

console.log(res);

//OUTPUT:

0:{value: 1}
1:{value: 2}
2:{value: 3}
```

4. **Chaining:**

```
let num=[1,2,-9,-8];

let res=num.filter((val)=>{val>=0}).map((val)=>{

   return {value:num}
});
console.log(res);
```

3. Filter

The filter() method creates a new array with elements that pass a given test (return true in the callback).

Syntax:

array.filter(callbackFunction(value, index, array));

Examples:

1. **Filter Even Numbers**:

```
let nums = [1, 2, 3, 4, 5, 6, 7, 8];

let evenNums = nums.filter((val) => {
        return val % 2 === 0
        });
console.log(evenNums); // Output: [2, 4, 6, 8]
```

2. **Filter Strings Longer Than 3 Characters**:

```
let strings = ["a", "abc", "abcd"];

let longStrings = strings.filter((str) => str.length > 3);
console.log(longStrings); // Output: ["abcd"]
```

4. Reduce

The reduce() method reduces an array to a single value by executing a callback on each element, accumulating a result.

Syntax:

array.reduce(callbackFunction(accumulator, currentValue, index, array), initialValue);

- **accumulator**: The value accumulated across iterations.
- **currentValue**: The current element being processed.
- **initialValue** (optional): The starting value for the accumulator.

Examples:

1. **Sum of an Array**:

```
let nums = [2, 3, 4, 5];

let sum = nums.reduce((res, currVal) => {
        return res + currVal;
        });
console.log(sum); // Output: 14
```

2. **Find Maximum Value**:

```
let max = nums.reduce((prev, curr) => (prev > curr ? prev : curr));
console.log(max); // Output: 5
```

3. **Product of an Array**:

let product = nums.reduce((acc, curr) => acc * curr, 1); //1 is intial value of acc
console.log(product); // Output: 120

Starter Code

- **<style>** tag connects HTML with CSS inside html file
- <link rel="stylesheet" href="style.css"/> ->connects to a css file

- <script > tag connects HTML with JS file

<script src="tut10-ImpArrayMethods.js"></script>

JavaScript: `try`, `catch`, `throw`

JavaScript provides a way to **handle errors** gracefully using `try`, `catch`, and `throw`.

These are used to:

- Prevent the program from crashing
- Provide user-friendly error messages
- Handle specific error cases

Syntax

```
try {
  // Code that may throw an error
} catch (error) {
  // Code that runs if an error occurs
} finally {
  // (Optional) Code that always runs
}
```

`throw`

- Used to **manually trigger an error**
- Can throw any value: string, number, object, or an `Error`

```
throw "Something went wrong!";
throw new Error("Custom error message");
```

Example 1: Simple try-catch

```
try {
  let x = y + 1; // y is not defined
} catch (err) {
  console.log("Caught an error:", err.message);
}
```

Example 2: Using `throw`

```
function divide(a, b) {
  if (b === 0) {
    throw new Error("Can't divide by zero");
  }
  return a / b;
}

try {
  console.log(divide(10, 0));
} catch (err) {
  console.log("Error:", err.message);
}
```

Example 3: `finally` block

```
try {
  console.log("Trying...");
  throw "Oops!";
} catch (err) {
  console.log("Caught:", err);
} finally {
  console.log("This runs no matter what.");
}
```

Window Object

The window object represents the browser's open window. It is **not part of JavaScript** but provided by the browser. It acts as a global object, and all its properties and methods are accessible globally.

Key Points:

- The window object contains properties like alert, document, setTimeout, and many more.(window.console.log(), window.alert())

Examples:

```
// Using window methods
window.alert("Hello!");   //   Same as alert("Hello!");
console.log("Hello!");    //  Same as window.console.log("Hello!");

// Accessing the window object in the console
console.log(window);
console.dir(window); // Lists all properties and methods of the `window` object
```

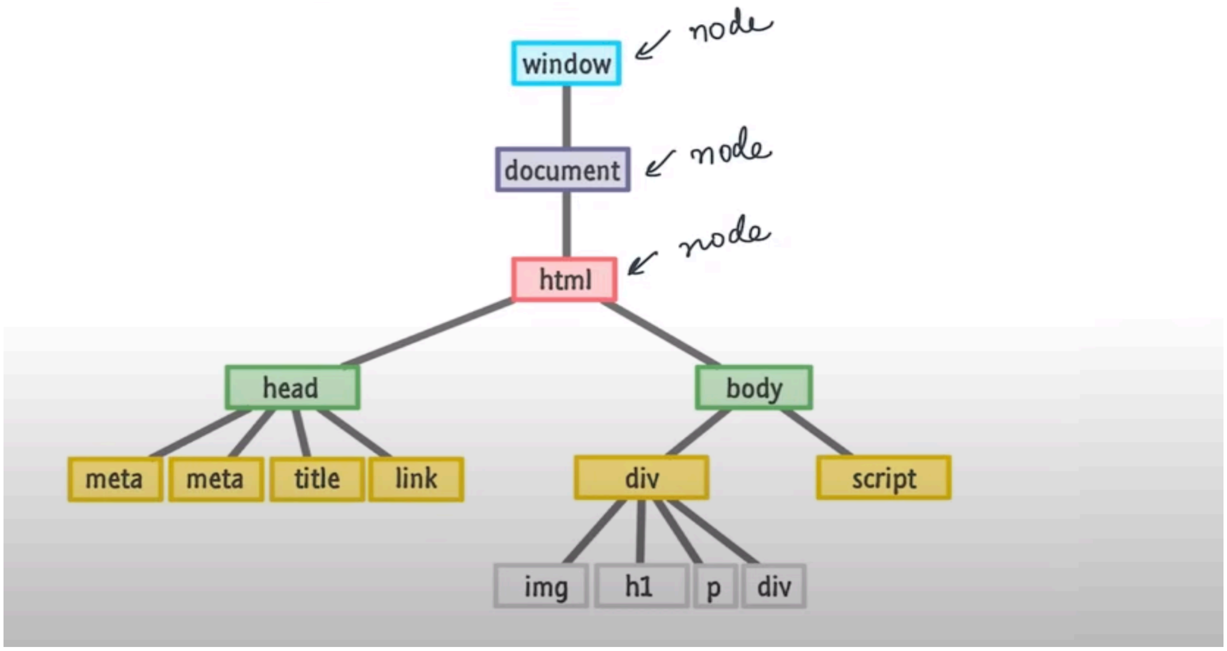
Note:

- **console.dir()** : is particularly useful when working with DOM elements or complex objects.
- It shows a **tree-like structure** of the object's properties in the browser's console.

Document Object Model (DOM)

The **DOM (Document Object Model)** represents the HTML code as a tree structure of objects in JavaScript.

- It allows **dynamic manipulation** of the website's content, style, and structure through JavaScript.
- The document object (a sub-object of window) gives access to the entire HTML structure.
- when a web page is loaded, the browser creates DOM of the page.



Accessing the DOM:

```
// Accessing the document object
console.log(document);          // Displays the document
console.dir(document);          // Lists all properties and methods of the document
```

```
// Accessing specific parts of the HTML
console.log(document.body);     // Returns the <body> element
console.log(document.head);     // Returns the <head> element
console.dir(document.head);     // Shows properties of the <head>
```

Example: Changing the background color

```
document.body.style.background = "lightblue";
```

Example: Changing the text

```
document.body.childNodes[3].innerText="Chaning Heading";
```

DOM Manipulation

HTML CODE:

```
<h1 id="heading1">JavaScript Course</h1>

<div id="div2" >
<h2 id="hiddenh2" style="visibility:hidden ;">Paris</h2>
<p class="city">Paris is the capital of France.</p>
<h2 class="city">India</h2>
<p class="city">Land of Culture</p>
</div>
```

1. Accessing Elements

(i) **By ID**: (.**getElementById(“”)**)

```
let heading = document.getElementById("heading1");
console.dir(heading); // Displays the properties and methods of the element
```

- **Note**: If the ID doesn't exist, it returns null.

(ii) **By Class(.getElementsByClassName(“”))**:

```
let cities = document.getElementsByClassName("city");
console.log(cities);   //Returns an HTMLCollection        (array-like object)
console.dir(cities);   // Access properties or iterate through it
```

- **Note**: If the class doesn't exist, it returns an empty HTMLCollection.

(iii) **By Tag Name(.getElementsByTagName(“”))**:

```
let paragraphs = document.getElementsByTagName("p");
console.log(paragraphs);  // Returns an HTMLCollection of all <p> elements
```

(iv) **Using Query Selector**: // **Query Selector**

`**querySelector**`: Returns the first element matching a CSS selector.

`**querySelectorAll**`: Returns all matching elements as a NodeList.

Syntax:

document.querySelector(" #ID / Tag / .class ");

document.querySelectorAll(" #ID / Tag / .class ");

*give id with “#” like : “#ID”

*give class with “.” like : “.class”

Notes:

- Both methods accept CSS selectors (e.g., `[type="text"]`).

- Throws `SYNTAX_ERR` if the selector is invalid.

```
// Select the first <p> element
let firstParagraph = document.querySelector("p");
console.dir(firstParagraph);

// Select all elements with the class "city"
let cityElements = document.querySelectorAll(".city");
console.dir(cityElements);

// Select all <p> tags
let allParagraphs = document.querySelectorAll("p");
console.dir(allParagraphs);

//select first and only element with id “myid”
let  sid=document.querySelector(“#myid”);
console.dir(sid);
```

NOTE: Additional feature inside a tag are called as Attributes.
Eg: <h1 id="hd1"> Hello </h1> //id is a n attribute

2. Properties:

NOTE: we have to select single element to access it therefore using .querySelectorALL() will give error: ‘undefined’.

(i) tagName

Returns the tag name of an element in uppercase.

Example:

```
<button id="myB">Click Me</button>


<script>
  let tg = document.querySelector("button"); // Select the button
  console.log(tg.tagName); // Output: 'BUTTON'
</script>
```

(ii) innerText

Returns the **visible text** content of an element, excluding hidden elements.

- **Example: Get innerText**

```
<div id="myDiv">Hello <b>World</b></div>

<script>
  let div = document.querySelector("#myDiv");
  console.log(div.innerText); // Output: "Hello World"
</script>
```

- **Example: Set innerText**

```
<script>
  div.innerText = "abcd"; // Replace the content inside the div
  console.log(div.innerText); // Output: "abcd"
</script>
```

(iii) innerHTML

Returns or sets the **HTML content** inside an element.

- **Example: Get innerHTML**

```
<div id="myDiv" class=”c1” >Hello <b>World</b></div>

<script>
```

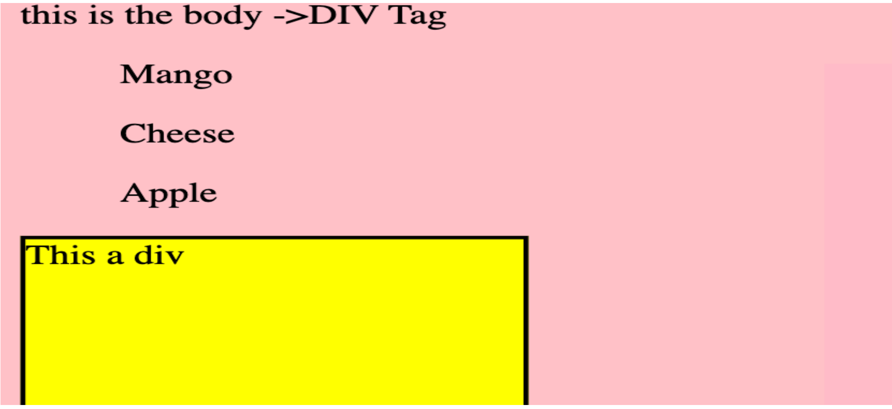
```
  let div = document.querySelector(".c1");
  console.log(div.innerHTML); // Output: "Hello <b>World</b>"
</script>
```

- **Example: Set innerHTML**

```
<script>
  // Replace content with a new div
  div.innerHTML = "<div>inner div</div>";
  console.log(div.innerHTML); // Output: "<div>inner div</div>"

  // Set italic text
  div.innerHTML = "<i>abcd</i>";
  console.log(div.innerHTML); // Output: "<i>abcd</i>"
</script>
```

Before Modifying



this is the body ->DIV Tag

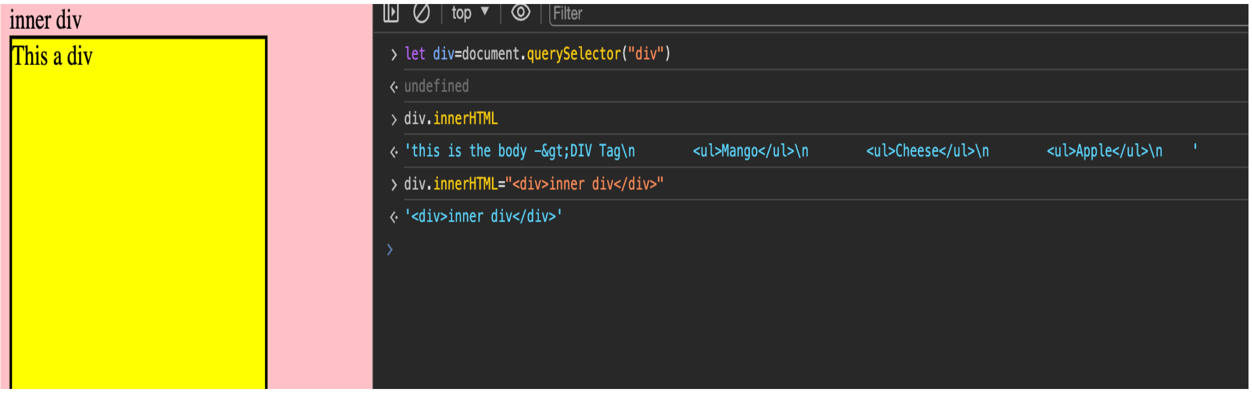
Mango

Cheese

Apple

This a div

After Modifying



inner div

This a div

```
> let div=document.querySelector("div")
<· undefined
> div.innerHTML
<· 'this is the body -&gt;DIV Tag\n        <ul>Mango</ul>\n        <ul>Cheese</ul>\n        <ul>Apple</ul>\n    '
> div.innerHTML="<div>inner div</div>"
<· '<div>inner div</div>'
>
```

(iv) textContent

Returns the **textual content**, including hidden elements. It does not include HTML tags.

- **Example:**

```
<h2 id="hiddenh2" style="visibility:hidden ;”>Hidden Heading</h2>


<script>
  let hiddenHeading = document.querySelector("#hiddenh2");

  console.log(hiddenHeading.innerText);    // Output: "" (because it's hidden)
  console.log(hiddenHeading.textContent); // Output: "Hidden Heading" (visible in script)
</script>
```

3. Attributes

(i) getAttribute(attr):

Used to get the value of a specified attribute (e.g., id, class, name, style, align).

Examples:

```
let button = document.querySelector("#myB"); // Select the button
console.log(button.getAttribute("id")); // Output: "myB"

let heading = document.querySelector("h2"); // Select the heading
console.log(heading.getAttribute("style")); // Output: The inline CSS style
```

(ii) setAttribute(attr, value):

Used to set or modify the value of an attribute.

Examples:

```
// Change the class name of a paragraph
let para = document.querySelector("p");
console.log(para.getAttribute("class")); // Output: "city"

para.setAttribute("class", "newClass");
console.log(para.getAttribute("class")); // Output: "newClass"

// Change the ID of a button
let button=document.querySelector("#myB")
console.log(button.getAttribute("id"));// Button1

button.setAttribute("id", "Bt");
console.log(button.getAttribute("id")); // Output: "Bt"
```

(iii) node.style:

Used to set inline styles on an element.

CSS Properties in JS:

- background-color → backgroundColor
- font-size → fontSize

Examples:

```
let div = document.querySelector("#box");

div.style.backgroundColor = "blue"; // Change background color

div.style.fontSize = "34px"; // Change font size

div.innerText = "This is a new box"; // Change text
```

4. Insert Elements

Steps:

1. **Create the element:** document.createElement(" tagName ")
2. **Add the element:** Select an element to add the new element (inside or outside) using append(), prepend(), before(), or after().

node.append(el) : adds at the end of node (inside)
node.prepend(el) : adds at the start of node (inside)
node.before(el) : adds before the node (outside)
node.after(el) : adds after the node (outside)

Examples:

(i) Create and Add a Button

```
let newBtn = document.createElement("button"); // Create button
newBtn.innerText = "Click Me"; // Set button text

let div = document.querySelector("#div2"); // Select a parent element
div.append(newBtn); // Add button to the end of div
```

(ii) Add a Heading

```
let newHeading = document.createElement("h1"); // Create heading
newHeading.innerHTML = "<i>Awesome Heading</i>"; // Set HTML content

document.querySelector("body").prepend(newHeading); // Add heading at the start
```

(iii)_ Append Child

appendChild(): adds a node to the end of the list of children of a specified parent node.

Example:

```
const parent = document.querySelector("button");

const newChild = document.createElement("p");
newChild.innerText = "Appended using querySelector!";

parent.appendChild(newChild);
console.log(parent);
```

5. Delete Elements

i) remove(): Removes an element completely from the DOM.

```
let para = document.querySelector("p"); // Select paragraph
para.remove(); // Remove the paragraph
```

(ii) removeChild(childNode): Removes a specified child element from the parent.

```
let parentNode = document.querySelector("#div2"); // Select parent
let childNode = parentNode.querySelector("p"); // Select child (first paragraph)
parentNode.removeChild(childNode); // // Removes <p> from parent
```

classList

The classList property is a read-only property that returns a **live DOMTokenList** of the class attribute and provides methods to add, remove, toggle, or check for classes.

It provides **methods to manipulate** the list of CSS classes of an HTML element

Although classList itself is read-only, you can modify its associated DOMTokenList using the following methods:

- **add(className)**: Adds a new class without removing the existing ones.
- **remove(className)**: Removes a class.
- **toggle(className)**: Adds the class if it's not already there, removes it if it's present.
- **contains(className))**: Returns true if the class is present, false otherwise.
- **replace(className))**: Replaces an old class with a new class.

Examples

1. Adding and Removing Classes:

```
const div = document.createElement("div");
div.className = "foo";

// Initial state: <div class="foo"></div>
console.log(div.outerHTML); // <div class="foo"></div>

// Remove the "foo" class and add a new class "another-class"
div.classList.remove("foo");
div.classList.add("another-class");

// Updated state: <div class="another-class"></div>
console.log(div.outerHTML); // <div class="another-class"></div>
```

2. Toggling Classes:

```
// Toggle the "visible" class (adds if not present, removes if already present)
div.classList.toggle("visible");

// Conditional toggle: adds "visible" if i < 10, otherwise removes it
let i = 5;
div.classList.toggle("visible", i < 10); // Adds "visible" as i is less than 10
```

3. Checking for Class Presence:

```
console.log(div.classList.contains("foo")); // false, as "foo" is removed
```

4. Adding/Removing Multiple Classes:

```
// Adding multiple classes
div.classList.add("foo", "bar", "baz");

// Removing multiple classes
div.classList.remove("foo", "bar", "baz");

// Using spread syntax to add/remove multiple classes
const cls = ["foo", "bar"];
div.classList.add(...cls);
div.classList.remove(...cls);
```

5. Replacing a Class:

```
// Replaces "foo" with "bar"
div.classList.replace("foo", "bar");
```

Practice Question

Qs. Create a <p> tag in html, give it a class & some styling.
Now create a new class in CSS and try to append this class to the <p> element.
Did you notice, how you overwrite the class name when you add a new one?

Solve this problem using classList.

```
let para=document.querySelector(".content");

 para.setAttribute("class","newClass");
```

//**Observation:** When using setAttribute to change the class, it completely **overwrites** the existing class, which results in the loss of previous styles or behaviors associated with it.

```
// Solution:

let para = document.querySelector(".content");
console.log(para.classList); // Output: DOMTokenList [ "existingClass" ]

// Add new class without removing the existing one
para.classList.add("newClass"); // Adds 'newClass' to existing list

// Output: DOMTokenList [ "existingClass", "newClass" ]

// Remove the new class
para.classList.remove("newClass"); // Removes 'newClass'
```

Qs. Access all div elements within the class name “box” and update their text content to something unique.

```
<body>
   <h2 id="h2">Hello Js</h2>
    <div class="box">First Div</div>
    <div class="box">Sec Div</div>
    <div class="box">Third Dhiv</div>
</body>
```

Sol.

1. **Manual Update Using Index**

```
let divs = document.querySelectorAll(".box");

divs[0].innerText = "Unique box 1";
divs[1].innerText = "Unique box 2";
divs[2].innerText = "Unique box 3";
```

2. **Using a Loop**

```
let divs = document.querySelectorAll(".box");
let i = 0;

for (let val of divs) {
   val.innerText = `Changed text ${i}`;
   i++;
}
```

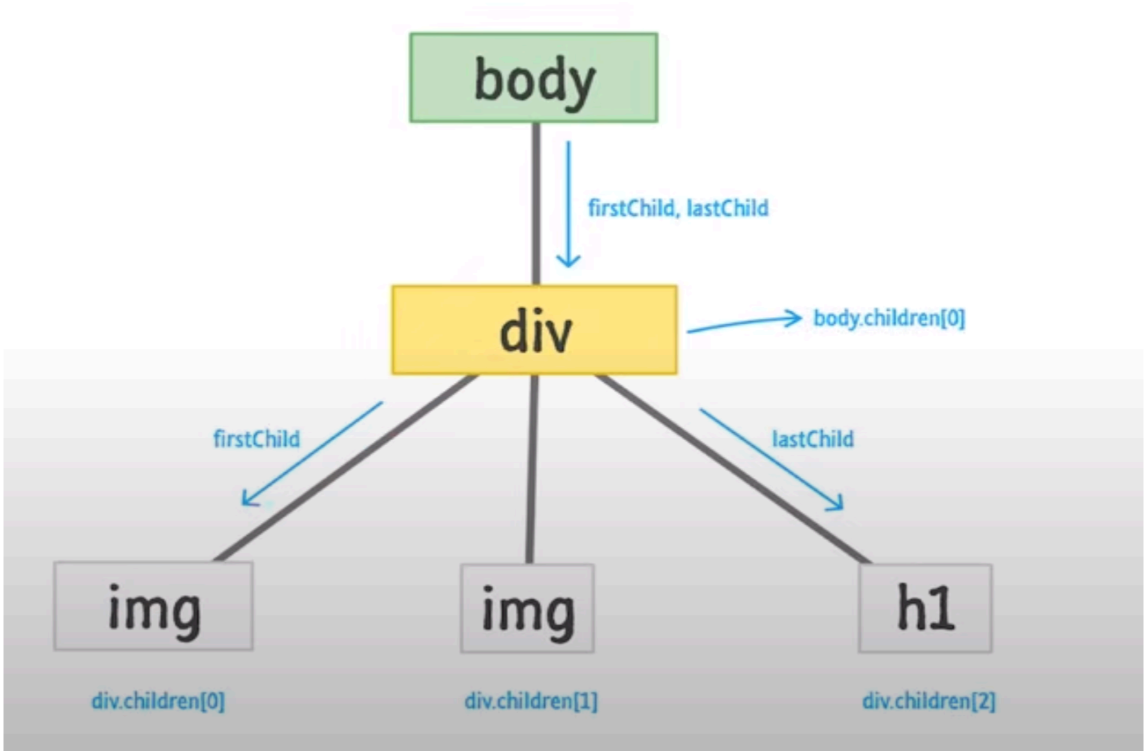
DOM Tree Structure

The DOM (Document Object Model) represents the structure of an HTML document as a hierarchical tree. Each element, attribute, or piece of text is a **node** in the tree.

Key Concepts

1. **Parent Node**: The node directly above another node.
   Example: <body> is the parent of <div>.
2. **Child Node**: Nodes directly nested under another node.
   Example: <div> is a child of <body>.
3. **Sibling Nodes**: Nodes that share the same parent.

Example: Two <img> elements and <h> inside the same <div> are siblings.



Types of Nodes in DOM Structures:

Here we are discussing three of the most important nodes:

1. Element Node (nodeType: 1)

1. Represents HTML tags like <div>, <p>, <span>, etc.
2. Can have attributes, child nodes (elements, text, or comments).

Example:

```
<div id="container">Hello</div>
```

<div> is an **element node**.

```
let element = document.getElementById("container");
console.log(element.nodeType); // Output: 1
```

NOTE:the values (text content) inside HTML tags are considered **child nodes** in the DOM, specifically **text nodes**.

2. Text Node (nodeType: 3)

1. Represents the textual content inside an element.

2. Always a child of an element node.

Example:

```
<p>This is text</p>
```

This is text is a **text node**.

```
let paragraph = document.querySelector("p");
console.log(paragraph.firstChild.nodeType); // Output: 3
console.log(paragraph.firstChild.nodeValue); // Output: "This is text”
```

NOTE:<p>This is text</p>: The text "This is text" is not an HTML element; it is a **text node**.
The **text node** is considered a child of the <p> element.

3. Comment Node (nodeType: 8)

1. Represents HTML comments, e.g., <!-- Comment -->.

2. Not visible in the browser but accessible in the DOM.

Example:

```
<!-- This is a comment -->
<h1>Hello</h1>
```

<!-- This is a comment --> is a **comment node**.

```
let comment = document.createComment("Sample comment");
console.log(comment.nodeType); // Output: 8
console.log(comment.nodeValue); // Output: "Sample comment"
```

4. Non-element Node

1. **Text Nodes** (nodeType: 3): Represent the textual content within elements.
2. **Comment Nodes** (nodeType: 8): Represent comments in the document.
3. **Document Node** (nodeType: 9):

- Represents the entire HTML document.
- It is the root node of the DOM tree.

Example:

```
console.log(document.nodeType); // Output: 9
```

4. **Document Fragment Nodes** (nodeType: 11):

- Represents a lightweight, minimal document object that is not part of the DOM.
- Useful for temporary DOM manipulation.

Example:

```
let fragment = document.createDocumentFragment();
console.log(fragment.nodeType); // Output: 11
```

Types of Property in DOM Structures:

1.parentNode Property

The parentNode property provides the parent of a specified node in the DOM tree.

- **Key Points**:
    - It returns null if the node has no parent, such as Document and DocumentFragment nodes.
    - If a node is not yet attached to the DOM, its parentNode will also be null.
    - Use parentElement if you only need the parent and want to exclude non-element nodes.
- **Syntax**: let parent = node.parentNode;
- **Example**: Removing a node from the DOM:

```
if (node.parentNode) {
  node.parentNode.removeChild(node);
}
```

2.childNodes Property

The childNodes property provides a live NodeList of all child nodes of a specified node.

- **Key Points**:

    - Includes all child nodes, such as **elements**, **text nodes**, and **comments**.
    - The list is **live**, meaning it updates automatically if child nodes are added or removed.
    - To get only element nodes, use children instead of childNodes.

- **Syntax**:

```
let children = node.childNodes;
```

- **Example**: Iterating over child nodes:

```
if (node.hasChildNodes()) {
  let children = node.childNodes;
  for (let child of children) {
    console.log(child.nodeName);
  }
}
```

3. firstChild Property

- Returns the first child node of a node. If the node has no children, it returns null.

- **Note**: This may return a **text node** (e.g., whitespace) or a **comment node** if they exist before any element node.

To specifically get the first element node, use **firstElementChild**.

- **Syntax**: **node.firstChild**

- **Example 1: With Whitespace**

```
<p id="para-01">
  <span>First span</span>
```

```
</p>

<script>
  const p = document.getElementById("para-01");
  console.log(p.firstChild.nodeName); // Output: "#text" (whitespace)
</script>
```

Explanation:

The space and newline characters between <p> and <span> are treated as a **text node** by the DOM.
Since firstChild refers to the first child node of <p>, it selects the **text node** created by the whitespace.

- **Example 2: Without Whitespace**

```
<p id="para-01"><span>First span</span></p>

<script>
  const p = document.getElementById("para-01");
  console.log(p.firstChild.nodeName); // Output: "SPAN"
</script>
```

- **Tip**: Use **firstElementChild** to get the first **element** node and avoid whitespace issues:

```
const firstElement = p.firstElementChild;
console.log(firstElement.nodeName); // Output: "SPAN"
```

4. lastChild Property

- **Definition**: Returns the last child node of a node. If the node has no children, it returns null.
- **Note**: Similar to firstChild, this may return a **text node** (e.g., whitespace) or a **comment node**.

Use **lastElementChild** to get only the last element node.

- **Syntax**: **node.lastChild**

- **Example**:

```
<tr id="row1">
  <td>First cell</td>
  <td>Last cell</td>
</tr>

<script>
  const tr = document.getElementById("row1");
  const lastChild = tr.lastChild;
  console.log(lastChild.nodeName); // Output may vary (e.g., "#text" if whitespace exists).
</script>
```

- **Tip**: Use lastElementChild to get the last **element** node:

```
const lastElement = tr.lastElementChild;
console.log(lastElement.nodeName); // Output: "TD"
```

5. nextSibling Property

- **Definition**: Returns the node immediately following the specified node within its parent's childNodes list.
- **Value**:
  - A Node object representing the next sibling.
  - null if the current node is the last child.
- **Note**:
  - It can return Text, Comment, or any type of node (including whitespace).
  - To skip non-element nodes (e.g., whitespace, comments), use nextElementSibling.

Example:

```
<div id="div-1">Here is div-1</div>
<div id="div-2">Here is div-2</div>
<br />
<output><em>Not calculated.</em></output>

<script>
  let el = document.getElementById("div-1").nextSibling;
  let i = 1;

  let result = "Siblings of div-1:\n";
  while (el) {
    result += `${i}. ${el.nodeName}\n`; // Log each sibling's nodeName
    el = el.nextSibling;
    i++;
  }

  document.querySelector("output").innerText = result;
</script>
```

Output:

```
Siblings of div-1:
1. #text
2. DIV
3. #text
4. BR
5. #text
6. OUTPUT
```

6. previousSibling Property

- **Definition**: Returns the node immediately preceding the specified node within its parent's childNodes list.

- **Value**:
  - A Node object representing the previous sibling.
  - null if the current node is the first child.
- **Note**:
  - Like nextSibling, it can return non-element nodes.
  - To get the previous element node, use previousElementSibling.

Example 1 (Without Whitespace):

```
<img id="b0" /><img id="b1" /><img id="b2" />

<script>
  console.log(document.getElementById("b1").previousSibling); // <img id="b0">
  console.log(document.getElementById("b2").previousSibling.id); // "b1"
</script>
```

Example 2 (With Whitespace):

```
<img id="b0" />
<img id="b1" />
<img id="b2" />

<script>
  console.log(document.getElementById("b1").previousSibling); // #text (whitespace node)
  console.log(document.getElementById("b1").previousSibling.previousSibling); // <img id="b0">
</script>
```

Events in JavaScript

- The change in the state of an object is called an event.
- Events are fired to notify code about "interesting changes" that may affect the execution of code.

Types of Events:

- **Mouse Events:** click, dblclick, mousemove, etc.
- **Keyboard Events:** keypress, keyup, keydown
- **Form Events:** submit, input, etc.
- **Touch Events:** touchstart, touchmove, etc.
- **CSS Animation and Transition Events:** animationstart, animationend, transitionend
- **Print Events** and more.

1. Click Event

- **Event Type:** click
- Fired when a user clicks on an element (button, link, image, etc.)

Example:

```
const button = document.querySelector("button");
button.addEventListener("click", () => {
   console.log("Button clicked!"); // Output: Button clicked!
});
```

- **When does it fire?** When a user clicks on an element with the click event listener.

2. Mouse Events

- **Event Types:** mousemove, mousedown, mouseup, mouseover, mouseout
- Handles mouse interactions:
    - **mousemove:** Fires when the mouse moves over an element.

    - **mousedown:** Fires when a mouse button is pressed.
    - **mouseup:** Fires when a mouse button is released.
    - **mouseover**: Fires when the mouse pointer enters an element.
    - **mouseout**: Fires when the mouse pointer leaves an element.

Example:

```
const div = document.querySelector("div");
div.addEventListener("mousemove", (event) => {
   console.log(`Mouse moved at X: ${event.clientX}, Y: ${event.clientY}`); // Output:
Mouse moved at X: <value>, Y: <value>
});
```

3. Keyboard Events

- **Event Types:** keydown, keyup
- Tracks keyboard interactions:
    - **keydown:** Fires when a key is pressed.
    - **keyup:** Fires when a key is released.

Example:

```
document.addEventListener("keydown", (event) => {
   console.log(`Key pressed: ${event.key}`); // Output: Key pressed: <key>
});
```

4. Input Events

- **Event Types:** input, change
- **input**: Fires when the value of an input element changes (e.g., typing in a text field).

- **change**: Fires when the user commits a change to an input element (e.g., when the input loses focus).

  Example:

```
const input = document.querySelector("input");
input.addEventListener("input", () => {
  console.log("Input changed"); // Output: Input changed
});
```

5. Focus Events

- **Event Types:** focus, blur
- **Description:**
  - **focus**: Fires when an element gains focus (e.g., when a text field is selected).
  - **blur**: Fires when an element loses focus (e.g., when a text field is deselected).

  Example:

```
const input = document.querySelector("input");
input.addEventListener("focus", () => {
  console.log("Input gained focus"); // Output: Input gained focus
});
```

6. Load Events

- **Event Types:** load, DOMContentLoaded
- **Description:**
  - **load**: Fires when the entire page and its resources (images, styles, etc.) are fully loaded.

    - **DOMContentLoaded**: Fires when the HTML has been fully parsed, without waiting for stylesheets or images.

Example:

```
window.addEventListener("load", () => {
  console.log("Page fully loaded"); // Output: Page fully loaded
});
```

7. Drag and Drop Events

- **Event Types:** dragstart, dragover, drop, dragend
- **Description:**
    - dragstart: Fires when dragging begins.
    - dragover: Fires when an element is being dragged over.
    - drop: Fires when the dragged element is dropped.
    - dragend: Fires when dragging ends.

Example:

```
const draggable = document.querySelector(".draggable");
draggable.addEventListener("dragstart", (event) => {
  console.log("Drag started"); // Output: Drag started
});
```

8. Animation Events

- **Event Types:** animationstart, animationend, animationiteration
- **Description:**
    - **animationstart**: Fires when an animation starts.
    - **animationend**: Fires when an animation ends.

  - **animationiteration**: Fires when an animation completes a cycle.

Example:

```
const element = document.querySelector(".animated");
element.addEventListener("animationend", () => {
   console.log("Animation ended"); // Output: Animation ended
});
```

9. Transition Events

- **Event Types:** transitionstart, transitionend
- **Description:**
  - transitionstart: Fires when a CSS transition starts.
  - transitionend: Fires when a CSS transition ends.

Example:

```
const element = document.querySelector(".transition");
element.addEventListener("transitionend", () => {
   console.log("Transition ended"); // Output: Transition ended
});
```

10. Touch Events

- **Event Types:** touchstart, touchmove, touchend
- **Description:**
  - **touchstart**: Fires when a touch point is placed on the screen.
  - **touchmove**: Fires when a touch point moves on the screen.
  - **touchend**: Fires when a touch point is removed from the screen.

Example:

```
document.addEventListener("touchstart", (event) => {
   console.log("Touch started"); // Output: Touch started
});
```

Event Handling:

Inline Event Handling

- **Not a good practice**: Mixing JavaScript with HTML.

html

- <button onclick="console.log('Button clicked'); alert('Hello')">Click here</button>
  //Print "button clicked" and alert message "hello" on clicking button.

- <button ondblclick="console.log('button 2 clicked')">click here 2 time</button>
  //Print "button 2 clicked" on clicking button.

- <div onmouseover="console.log('inside div1')">div 1</div>
  //Print "inside div1" when mouse hover above div1

JavaScript Event Handling (Better)-> writing in JS

- **Syntax:** node.event = () => {   /* handler */   }

(i)
```
let bt1 = document.querySelector("#bt1");
bt1.onclick = () => {
   console.log("Button 1 clicked");
};
```

(ii)
```
let box=document.querySelector("div");
box.onmouseover=()=>{
        console.log("Hovering in div1");
};
```

NOTE:

(i) ***Inline vs JavaScript Event Handling:** If an event is handled **both inline (in HTML)** and in **JavaScript**, the JavaScript event will **override** the inline event.*

(ii) ***Overwriting Event Handlers:** Assigning multiple handlers to the same event using node.event (e.g., node.onclick) **overwrites the previous handler**.*

Example:

```
let btn = document.querySelector("#btn");

// First handler
btn.onclick = () => {
   console.log("Handler 1");
};

// Second handler overwrites the first
Btn.onclick = () => {
   console.log("Handler 2");
};

// Output: Only "Handler 2" will run when the button is clicked.
```

*** This means we can only use a single handler for an event using node.event.
Therefore we use **<u>addEventListener</u> to a**void overwriting and assigning multiple handlers to same event. ***

Event Object:

- The event object contains details about the event.
- All event handlers have access to the Event Object's properties and methods.

- Syntax: **node.event = (e) => {**
  // handle here
  };

- **Properties:** e.target, e.type, e.clientX, e.clientY and many more.
  - target: The object that triggered the event.
  - type: The type of event (e.g., click).
  - clientX: The horizontal coordinate where the event occurred.
  - clientY: The vertical coordinate where the event occurred.

Example:

```
let bt1 = document.querySelector("#bt1");
bt1.onclick = (e) => {
   console.log(e.target);  // Logs the element triggering the event
   console.log(e.type);    // Logs the event type (click)
   console.log(e.clientX); // Logs the X-coordinate of the event
   console.log(e.clientY); // Logs the Y-coordinate of the event
};
```

Event Listeners:

- **Syntax:**

node.addEventListener(event, callback);
node.removeEventListener(event, callback); // Removes the listener

- **Multiple Handlers:** You can add multiple listeners to the same event.

```
bt1.addEventListener("click", () => {
   console.log("Handler 1");
});
bt1.addEventListener("click", () => {
   console.log("Handler 2");
});
```

- we can also add event object

```
let box=document.querySelector("div");

box.addEventListener("mouseover",(evt)=>{
console.log("inside div1");
console.log(evt);
console.log(evt.type);
});
```

- **Removing Event Listener:**
- Syntax**: node.removeEventListener(event, callback);**

The **removeEventListener** method requires the **exact same function reference** that was passed when the event listener was added.

```
(i). let bt1 = document.querySelector("#bt1"); // Adding multiple event listeners
      bt1.addEventListener("click", () => {
         console.log("button 1 was clicked -handler 1");
      });
      bt1.removeEventListener("click", () => {
         console.log("button 1 was clicked -Handler 2");
        });
//function is not removed because it is different from referenced(diff. storage in memory)
function even though the code is same.
```

```
(ii).
let bt1=document.querySelector("#bt1");

//stroring in variable
const handler1=()=>{
console.log("button 1 was clicked -Handler 1");
};
```

```
  //adding it in eventlistner
bt1.addEventListener("click",handler1);

//removing eventListner
bt1.removeEventListener("click",handler1)
```

Practice: Toggle Dark Mode

Objective: Create a button to toggle between dark and light mode.

Method 1:

```
let currMode = "light";
let tg = document.querySelector("#tg");

tg.addEventListener("click", () => {
    if (currMode == "light") {
        document.querySelector("body").style.backgroundColor = "black";
        currMode = "dark";
    } else {
        document.querySelector("body").style.backgroundColor = "white";
        currMode = "light";
    }
});
```

Method 2: Using classList.add and classList.remove:

```
let currMode = "light";

let tg = document.querySelector("#tg");
let body = document.querySelector("body");


tg.addEventListener("click", () => {
    if (currMode == "light") {
        body.classList.add("dark");  // Adds dark class
        body.classList.remove("light"); // Removes light class
        currMode = "dark";
    } else {
        body.classList.add("light");  // Adds light class
        body.classList.remove("dark"); // Removes dark class
        currMode = "light";
    }
});
```

Method 3:using classlist->toogle

```
tg.addEventListener("click", () => {
body.classList.toggle("dark");
body.classList.toggle("light");
});
```

Method 4: Using setAttribute:

```
tg.addEventListener("click", () => {
    if (currMode == "light") {
        body.setAttribute("class", "dark"); // Sets dark class
        currMode = "dark";
    } else {
        body.setAttribute("class", "light"); // Sets light class
        currMode = "light";
    }
});
```

JavaScript Notes: Objects, Prototypes, Classes, and Inheritance

Objects :

An object is an entity having having state and behavior (properties and method)

1. **Defining Methods Inside an Object**:

```
const student = {
   fullName: "Paras",
   marks: 49,
   printMarks1: function () {
      console.log("Marks of", this.fullName, "is", this.marks);
   },
   printMarks2() {
      console.log("Another way of writing a function inside an object");
   }
};
student.printMarks1(); // Output: Marks of Paras is 49
student.printMarks2(); // Output: Another way of writing a function inside an object
```

2. **Use of this Keyword**: Refers to the object that calls the method.

Example:

```
console.log(this.fullName); // Refers to fullName of the calling object
```

Prototypes in JavaScript

a. All JavaScript objects inherit properties and methods from their prototype.
b. A prototype is itself an object.

Example:

```
let arr = ["apple", "litchi", "orange"];
arr.push("mango"); // 'push' is part of the array prototype
```

Custom Prototypes: You can set or modify an object's prototype using **__proto__ :**

Example:

```
const employee = {
   calcTax() {
      console.log("Tax is 10%");
   }
};

const paras = { salary: 60000 };

paras.__proto__ = employee; // Set employee as prototype
paras.calcTax(); // Output: Tax is 10%
```

Note:

(i) Prototypes are just a refernce to an object

(ii) If an object and its prototype both have the same method, the **object's method** is executed.

Example:

```
const employee={
   calcTax(){

      console.log("Tax is 10%");

   }

};


const paras={

   salary:60000,

   calcTax(){      //same function as employee(that will be used as prototype)

      console.log("tax is 20%")
```

```
    },
};


paras.__proto__=employee

console.log(paras.calcTax)// tax is 20%
```

Classes in JavaScript

Classes are blueprints for creating objects that share common properties and behaviors.

Class Syntax:

```
class MyClass {
    constructor() { ... }
    myMethod() { ... }
}
```

Example:

(i)

```
class Car {
    constructor(brand) {
        this.brand = brand;
    }
    start() {
        console.log("Car starts");
    }
}

let toyota = new Car("Toyota");
toyota.start(); // Output: Car starts
```

Constructor:

Even if you don't define a constructor, JavaScript automatically creates one.Are called/created automatically at intialization of objects.

Example:

```
let toyota = new Car();

console.log(toyota);
// Output:
// Car {}
// [[Prototype]]: Object
// constructor: class Car //even though we  didn't define constructor
// setBrand: f setBrand(brand)
// start: f start()
// [[Prototype]]: Object
```

1. **Custom Constructor**:

   Used to initialize an object with specific values.

Example:

```
class Car {
   constructor(brand) {
      console.log("Creating a new object");
      this.brand = brand;
   }
}

let tata = new Car("Nexon");                    //brand=”Nexon”
// Output: Creating a new object
let toyota = new Car();        //brand=undefined
// Output: Creating a new object
```

2. **Multiple Parameters in Constructor**:

```
class Car {
    constructor(brand, mileage) {
        this.brand = brand;
        this.mileage = mileage;
    }
}
let tata = new Car("Punch", 10);
console.log(tata);
```

Inheritance in JavaScript

A child class can inherit properties and methods from a parent class using extends.

Syntax:

```
class Parent { ... }
class Child extends Parent { ... }
```

Example:

```
class Person {
 constructor(){
  this.speices="homo Sapiens Sapein";
 }
    eat() {
        console.log("Person eats");
    }
}

class Engineer extends Person {
    work() {
        console.log("Engineer solves problems");
    }
}
```

```
let eng = new Engineer();   //species:”homo sapiens sapiens”
eng.eat(); // Output: Person eats
eng.work(); // Output: Engineer solves problems
```

Method Overriding:If child and parent have the same method, the **child's method** is used.

super Keyword

To call the parent class's constructor or methods in the child class.

Example 1: Using super in Constructor:

```
class Person {
    constructor(name) {
        this.name = name;
    }
}

class Engineer extends Person {
    constructor(name, branch) {
        super(name); // Calls the parent class's constructor
        this.branch = branch;
    }
}

let eng = new Engineer("Param", "Computer");
console.log(eng); // Output: Engineer { name: 'Param', branch: 'Computer' }
```

Example 2: Error Without super:

```
class Person {
    constructor() {
        this.species = "Homo Sapiens";
    }
}
```

```
class Engineer extends Person {
   // ERROR: Must call super constructor in derived class before accessing 'this'
   constructor(branch) {
      this.branch = branch; // Causes an error
   }
}

// Corrected Version:
class Engineer extends Person {,
   constructor(branch) {
      super(); // Fixes the error
      this.branch = branch;
   }
}

let eng = new Engineer("Mechanical");
console.log(eng); // Output: Engineer { species: "Homo Sapiens", branch: "Mechanical" }
```

Example 3:

```
class Person{
   constructor(name){
        this.speices="homo Sapiens Sapein";
        this.name=name;
        }
};
class Engineer extends Person{ };

let p=new Person();    //Person{ species: "Homo Sapiens", name: undefined"}
 let eng=new Engineer("Param");    // Engineer { species: "Homo Sapiens", name:
"Param }
```

Note: If no constructor is defined by child class it inherits constructor of parent class

Example 4: Calling Parent Methods:

```
class Person {
   eat() {
      console.log("Person eats");
   }
}

class Engineer extends Person {
   work() {
      super.eat(); // Calls the parent class's method
      console.log("Engineer solves problems");
   }
}

let eng = new Engineer();
eng.work();
// Output:
// Person eats
// Engineer solves problems
```

Practice Questions

Question 1: Create a User class with properties name and email. Add a method viewData.

Question 2: Create an Admin class that extends User. Add a method editData.

Solution:

```
let data = "Web Data";

class User {
   constructor(name, email) {
      this.name = name;
      this.email = email;
   }
   viewData() {
      console.log("Data =", data);
   }
}

class Admin extends User {
   editData(newData) {
      data = newData;
   }
}

let admin = new Admin("Admin", "admin@example.com");
admin.editData("New Web Data");
console.log(data); // Output: New Web Data
```

Synchronous and Asynchronous Execution

1. **Synchronous Code**:Executes line by line, where each instruction waits for the previous one to finish.

Example:

```
console.log("1");
console.log("2");
```

Output:

```
1
2
```

2. **Asynchronous Code**:Allows certain tasks (like waiting for a timeout) to run in the background without blocking subsequent code.

Example:

```
console.log("One");
console.log("Two");

setTimeout(() => {
  console.log("Wait");
}, 4000);

console.log("Three");
console.log("Four");
```

Output:

```
One
Two
Three
Four
Wait (after 4 seconds)
```

Callbacks

A function passed as an argument to another function and executed later.

Example:

```
function sum(a, b) {
  console.log(a + b);
}
```

```
function calculator(a, b, callback) {
   callback(a, b);
}

calculator(4, 5, sum); //callback function is passed without argument
// Output: 9
```

1. **Inline Callback**: Define the callback function directly inside the function call:

```
calculator(4, 5, (a, b) => {
   console.log(a + b);
}); // Output: 9
```

2. **Callback Inside setTimeout()**:

```
function hello() {
   console.log("Hello");
}

setTimeout(hello, 2000); // Output: Hello (after 2 seconds)
```

Callback Hell

A situation where callbacks are nested deeply, leading to hard-to-read and manage code (Pyramid of Doom).

Example:

```
function getData(id, nextCallback) {
   setTimeout(() => {
      console.log("Data", id);
      if (nextCallback) {      // if there is a next callback function
```

```
            nextCallback();
        }
    }, 2000);
}
```

(i)

```
getData(1);
getData(2);
getData(3);
```

OUTPUT:

```
 2sec wait
Data 1
Data 2
Data 3
```

```
(ii)   GetData(1 , getData(2))
       //Invalid Syntax
```

(iii)***// Nested callbacks/Callback hell***

```
getData(1, () => {
    getData(2, () => {
        getData(3, () => {
            getData(4);
        });
    });
});
```

OUPTUT:

```
2sec wait
Data 1
2sec wait
Data 2
2sec wait
Data 3
2sec wait
```

Promises

Promise is for “**completion”** of tasks. It is an object in JS.

They are a solution to callback hell.

resolve and **reject** are callbacks provided by Promise

States of a Promise:

a. **Pending**: Initial state, neither resolved nor rejected.
b. **Fulfilled**: Operation completed successfully (resolve() called).
c. **Rejected**: Operation failed (reject() called).

Basic Syntax:

```
let promise = new Promise((resolve, reject) => {
   // Task logic here
});
```

Examples:

Pending State:

```
        let promise = new Promise((resolve, reject) => {
                console.log("I am a promise");
        });
        console.log(promise);
```

Output:

```
I am a promise
 Promise {<pending>}
 [[Prototype]]: Promise
 [[PromiseState]]: "pending"
 [[PromiseResult]: undefined
```

Resolved State:

```
let promise = new Promise((resolve, reject) => {
   resolve("Success");
});
console.log(promise);
```

Output:

```
* Promise {<fulfilled>: 'success'}
[[Prototype]]: Promise
[[PromiseState]]: "fulfilled"
[[PromiseResult]]: "success"
```

Rejected State:

```
let promise = new Promise((resolve, reject) => {
   reject("Failure");
});
console.log(promise);
```

Output:

```
Promise {<rejected>: 'I rejected this promise'}
[[Prototype]]: Promise
[[PromiseState]]: "rejected"
[[PromiseResult]]: "I rejected this promise"
Uncaught (in promise) I rejected this promise
```

Eg:

```
 function getDataID(id,nextCallback){

        return new Promise((resolve,reject)=>{

          setTimeout(() => {

          console.log(id);

          resolve("success");

          if (nextCallback) {

            nextCallback();

          }
```

```
  }, 2000);
  })
}

let promise=getDataID(123);
```

```
Promise // Promise {<pending>}
**After 2 sec**
123
Promise //Promise {<fulfilled>: 'success'}
```

.then() and .catch() in Promises

1. **.then()**: Defines what should happen if the promise is **fulfilled** (resolved).
2. **.catch()**: Handles errors when the promise is **rejected**.

Syntax:

```
promise
  .then((result) => {
    // Code to handle success
  })
  .catch((error) => {
    // Code to handle errors
  });
```

Example: Handling a Promise

```
const getPromise = () => {
  return new Promise((resolve, reject) => {
    console.log("hello there");

     resolve("Success");

  });
};

let promise = getPromise();

promise
  .then((res) => {
    console.log("Promise fulfilled:", res); // This runs when resolved
  })
  .catch((err) => {
    console.log("Promise rejected:", err); // This runs when rejected
  });
```

Output (if “resolve")is used):

```
hello there
Promise fulfilled: Success
```

Promise Chaining

Example 1:

```
function async1() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('Data 1');
      resolve("Success");
    }, 2000);
  });
}

function async2() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('Data 2');
      resolve("Success");
    }, 2000);
  });
}
```

(i)Asynchronous

```
let p1 = async1();
p1.then((res) => {
     console.log(res);
  });
let p2 = async2();
p2.then((res) => {
   console.log(res);

});
```

Output:

Both asynchronous tasks (async1 and async2) will execute simultaneously:

```
2sec wait
Data 1
Data 2
Success
Success
```

(ii) Sequential Execution with Chaining(synchronous)

Syntax1:

```
async1().then((res)=>{
```

```
          async2().then((res)=>{
                    console.log(res);
                    });
          });
```

Syntax2:

```
   async1()
            .then((res) => {
              console.log(res);
              return async2();  // Returning the promise from async2
            })
            .then((res) => {
              console.log(res);
            });
```

Output:

The second task (async2) starts **only after** the first task (async1) is completed:

```
Data 1
Success
2sec wait
Data 2
Success
```

Example 2:

```
function getData(id) {

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      console.log("data", id);

      resolve("Great Success");
    }, 4000);

  });

}
```

Promise Chaining

Syntax 1:

```
getData(1).then((res)=>{
    getData(2).then((res)=>{
            getData(3).then((res)=>{
                console.log(res);
                      })
})
})
```

Syntax 2:

```
// Using chaining for sequential execution

getData(1)

  .then((res) => {

    return getData(2);

  })

  .then((res) => {

    return getData(3);

  })

  .then((res) => {

    console.log(res);

  });
```

Output:

Each task waits for the previous one to finish:

```
4sec wait
data 1
4sec
data 2
4sec
data 3
Great Success
```

Async-Await

1. **Async Functions**:
   Always return a Promise.
   Syntax: async function myFunc() { ... }.

2. **Await**:

   Pauses the execution of the surrounding async function until the promise is resolved or rejected.
   Can **only** be used inside an async function or the top-level body of a module.

Basic Syntax:

```
async function myFunction() {
   let result = await someAsyncOperation();
   console.log(result);
}
```

Examples

Example 1: Basic Async Function

```
async function hello() {
  console.log("hello");
}

console.log(hello());
```

Output:

```
hello
Promise {<fulfilled>: undefined}
```

- The hello() function does not return a promise explicitly, but async automatically wraps the function in a resolved promise.

Example 2: Using await

```
function api() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("weather data");
      resolve(200);
    }, 2000);
  });
}

async function getWeatherData() {
  console.log("Fetching first set of weather data...");
  await api(); // Pauses here until the promise is resolved
  console.log("Fetching second set of weather data...");
  await api();
}
```

```
console.log(getWeatherData());
```

Output:

```
Fetching first set of weather data…
2sec
weather data
Fetching second set of weather data…
2sec
weather data
Promise {<fulfilled>: undefined}
```

- getWeatherData() runs sequentially, waiting for each api() call to resolve before moving on.

Example 3: Fetching Sequential Data

```
function getData(id) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("data", id);
      resolve("Great Success");
    }, 4000);
  });
}

async function getAllData() {
  console.log("Getting data1...");
  await getData(1);
  console.log("Getting data2...");
  await getData(2);
  console.log("Getting data3...");
  await getData(3);
  console.log("Getting data4...");
  await getData(4);
}
```

```
console.log(getAllData());
```

Output:

```
Getting data1...
data 1
Getting data2...
data 2
Getting data3...
data 3
Getting data4...
data 4
Promise {<fulfilled>: undefined}
```

- The getAllData() function ensures data is fetched sequentially using await.

Benefits of Async-Await

1. **Improved Readability**: Resembles synchronous code, avoiding deeply nested .then() chains.
2. **Sequential and Controlled Execution**: Makes it easier to manage tasks that depend on the previous one.
3. **Error Handling**: Combine try-catch blocks with async-await for cleaner error management.

Error Handling with Async-Await

```
function faultyApi() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      reject("Failed to fetch data");
    }, 2000);
  });
}
```

```
async function fetchData() {
  try {
    await faultyApi();
    console.log("Data fetched successfully");
  } catch (error) {
    console.log("Error:", error);
  }
}

fetchData();
```

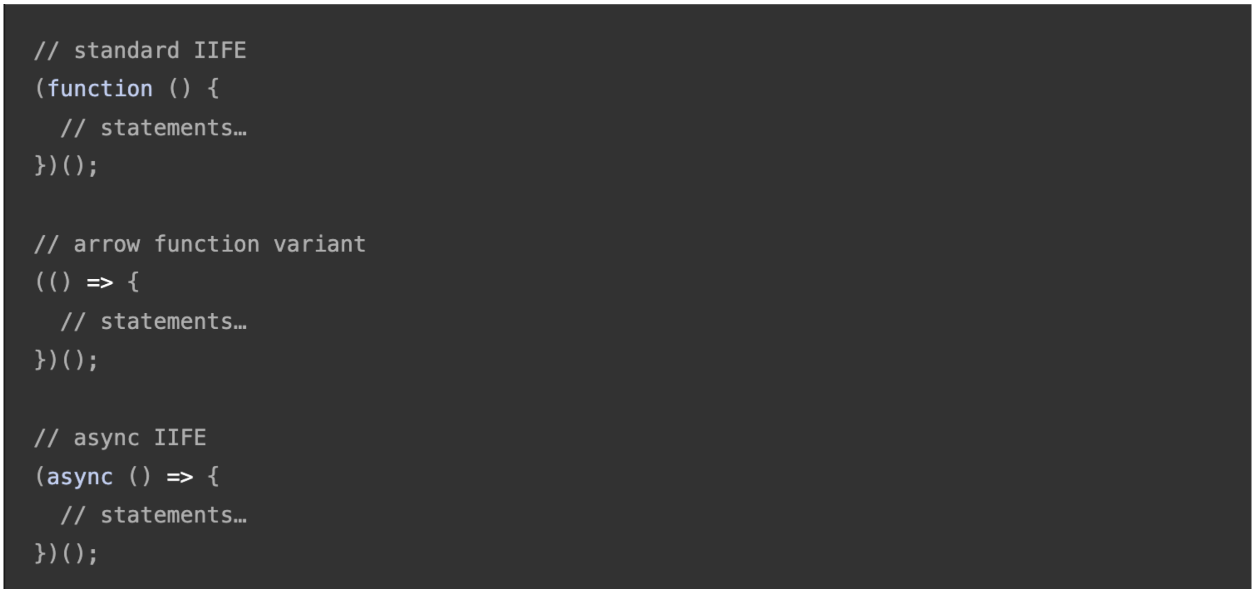
Output:

```
Error: Failed to fetch data
```

IIFE (Immediately Invoked Function Expression)

A function that runs immediately after it is defined.

Syntax:



```
// standard IIFE
(function () {
  // statements…
})();

// arrow function variant
(() => {
  // statements…
})();

// async IIFE
(async () => {
  // statements…
})();
```

Use cases of IIFEs include:

- Avoiding polluting the global namespace by creating a new scope.
- Creating a new async context to use await in a non-async context.
- Computing values with complex logic, such as using multiple statements as a single expression.

Example with async:

```
(async function () {
   console.log("Getting Data 1...");
   await fetchData(1);

   console.log("Getting Data 2...");
   await fetchData(2);

   console.log("Getting Data 3...");
   await fetchData(3);
})();
```