JavaScript

- JS is a programming language. We use it to give instruction to the computer.

  The key characteristics of JavaScript in short points:

  1. Interpreted Language
  2. High-level Language
  3. Dynamic Typing
  4. Prototype-based
  5. Functional and Object-Oriented
  6. Event-driven and Asynchronous
  7. Client-side scripting
  8. Cross-platform
  9. Rich Ecosystem
  10. Versatile Usage

#Variables

- Container for data of name of memory location where data is stored.

- Rules for naming variables:-

  (i) Variables names are case-sensitive.

  (ii)Only letters,digits,underscore(_) and $ is allowed(not even space)

  (iii)Only a letter, underscore(_) or $ can be 1st character.

  (iv)Reserved words cannot be used .

#let , var & const

- var :- Variable can be re-declared and updated. A global scope variable.
- let :- Variable cannot be re-declared but can be updated. A block scope variable.
- const :- Variable cannot be re-declared or updated, A block scope variable.

#DataTypes in JS

1. Objects such as Arrays , Functions etc..

```
let obj ={ name:"tej", roll:23 };
console.log(obj.name); // Output: "tej"
```

jstej

```
console.log(obj.roll); // Output: 23
console.log(obj['name']); // Output: "tej"
console.log(obj['roll']); // Output: 23
alert("success");
```

2. Primitive datatypes :-

   (i) Number Represents numeric data. It includes integers, floating-point numbers, and special numeric values like NaN (Not a Number) and Infinity.
   Example:

   ```
   let x = 10;
   ```

   (ii)String Represents textual data enclosed within single or double quotes. It can contain letters, numbers, symbols, and whitespace.
   Example:

   ```
   let name = "John Doe";
   ```

   (iii)Boolean Represents a logical value indicating true or false.
   Example:

   ```
   let isReady = true;
   ```

   (iv)UndefinedRepresents a variable that has been declared but not assigned a value.
   Example:

   ```
   let y; // y is undefined
   ```

   (v)Null Represents an intentional absence of any value.
   Example:

   ```
   let age = null;
   ```

   (vi)BigInt Represents integers with arbitrary precision, allowing for the storage of numbers beyond the limit of the Number type.
   Example:

   ```
   const bigNumber = 1234567890123456789012345678901234567890n;
   ```

   (vii)Symbol Represents a unique identifier that's not visible to code outside of the function or block in which it's defined.

   const mySymbol = Symbol("description");

   "you can use typeof () function to get datatype of variable".

#Comments

- Part of code that is not executed it is basically used to explain code logic and make the code more understandable. Comments are of two types "Single line" and "Multi Line".

  ```
  //This is a single line comment
  /* this is a multiline
  comment */
  ```

#Operators

- Used to perform various mathematical operations on operands.
    1. Arithmetic Operators:-
        - + , - , * , / , %(modulus) , **(exponentiation) , ++(post and pre increment unary operators) ,—(post and pre decrement)
    2. Assignment Operators:-
        - = , += , -= , *= , %= , **=
    3. Comparison Operators:-
        - == , ===(checks type also) , ! = , ! ==, > , > = , < , < =
    4. Logical Operators :-
        - &&(Logical AND) , || (Logical OR) , ! (Logical NOT)
    5. Bitwise Operators:-
        - &(and) , |(or) , ^(x-or) , ~(not) , <<(left shift) , >>(right shift)
    6. Ternary Operator
        - used to check for conditions in a more cleaner way.

        ```
        age > 18 ? "adult" : "not adult";
        ```

#Conditional Statements

- To handle conditional logic.

1. if Statement

    ```
    let color;
    if(mode === "dark-mode"){
        color = "black";
    }
    ```

2. if else Statement

    ```
    let color;
    if(mode === "dark-mode"){
        color = "black";
    }
    else{
       color="white"p
    }
    ```

3. id elseif else Statement

    ```
    let color;
    if(mode === "dark-mode"){
        color = "black";
    }
    else if{
       color="white"p
    }
    else{
       console.log("sorry");
    }
    ```

4. Break Statement :- To move out from the loop or exit the loop.
5. Continue Statement :- To skip current iteration and move to next.

jstej

```
for (let i=0 ; i<=7;i++){
  if(i==5){
     continue;
  }
  console.log(i);
  }
```

6. Switch Statement

```
number = prompt("Enter a number");
switch (number) {
    case 1:
       console.log("number is 1");
       break;
    case 2:
        console.log("Number is two");
        break;
    default:
        console.log("other number");
        break;
}
```

#Loops

- Used to perform a particular iteration again and again without repeating or writing same code multiple times.

1. for Loop:-

```
let sum =0;
for(let i = 0 ;i<=5;i++){
   sum+=i;
}
console.log(sum);
```

2. while loop:-

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

3. do while loop :-

```
i=0;
do{
   console.log(i);
   i++;
 }while(i<=5);
```

4. infinite loop :-

```
for(let i=1;i>=0;i++){
  console.log("hy");
}
```

5. for-of loop :- To iterate over various sequences and objects.

```
name="tej"
for(let val of name){
   console.log(val);
}
```

6. for-in loop:- To iterate over objects.

```
names=["tej","amit"];
for(let val in names){
   console.log(val);
}
```

#Strings

- String is a sequence of characters used to represent text.

```
//string creation
let str = "tej";
//printing length of string
console.log(str.length);
//using indices
console.log(str[0]);
```

- String Interpolation (Template Literals)

```
let pen ="blue";
console.log(`use ${pen} pen`);
```

- String Methods(in-built functions to manipulate strings)
  - str.toUpperCase()
  - str.toLowerCase()
  - str.trim() :- removes whitespaces
  - str.slice(start,end?) :- returns part of string
  - str1.concat(str2);
  - str.charAt(idx);
  - str.replace(searchVal,newVal);

#Arrays

- Collection of items of same type

```
//create Array
let heros = ["ironman" ,"hulk","thor","batman"];
//indices
console.log(heros[0]);
//looping over
for(let val in heros){
  console.log(val);
}
```

- Arrays Methods.
  - push() : add element to the end
  - pop() : delete from end and return deleted element
  - toString() : converts array to string
  - concat() : joins multiple arrays and retruns result;
  - unshift() : adds element to start
  - shift() : delete from start and return
  - slice(startIdx,endIdx): returns a piece of array
  - splice(startIdx,delCount,newElement1,......)

  ```
  let fruits = ["Banana", "Orange", "Apple", "Mango"];

  // Remove 1 element at index 2
  fruits.splice(2, 1);
  // Result: ["Banana", "Orange", "Mango"]

  // Add elements at index 2
  fruits.splice(2, 0, "Lemon", "Kiwi");
  // Result: ["Banana", "Orange", "Lemon", "Kiwi", "Mango"]

  // Replace 1 element at index 0
  fruits.splice(0, 1, "Peach");
  // Result: ["Peach", "Orange", "Lemon", "Kiwi", "Mango"]
  ```

#Functions

- Block of code that can be reused again and again to perform a particular task .

```
//function definition
//parameters inside functions are blocked scope
function calSum(a,b){
    return a+b;
}
function printhelo(){
  console.log("hello");
}
//arrow function
const arroSum=(a,b)=>{
   return a+b;
};
//function calling
let result = calSum(5,6);
console.log(result);
printhelo();
```

#forEach Loop

A callback function is passed as an argument

```
arr=[1,2,3,4,5];
arr.forEach((val)=>{
  console.log(val);
});

//sqaure of each number
arr.forEach((val)=>{
  console.log(val*val);
});
```

Difference between functions and methods

- Functions are used to perform specific task or used to generate output on the basics of received input
- A method is a function that is associated with a particular object such map method associated with array.

#Map Method

- Creates a new array with the results of some operation. The value its callback returns are used to form new array.

```
oldarr = [1,2,3,4,5,6];
let newArr =  oldarr.map((val)=>{
  return val*2;
})

newArr
```

#filter Method

- creates a new array of elements that give true for a condition

```
arr=[1,2,3,4,8,44];
newArr=arr.filter((val)=>{
   return val %2===0;
})
console.log(newArr);
```

#Reduce Method

- Performs some operations and reduces the array to a single value. It returns that single value

```
arr=[1,2,3,4];
const output = arr.reduce((res,curr)=>{
   return res+curr;
});
console.log(output);

const output2 = arr.reduce((prev,curr)=>{
   return prev>curr?prev:curr;
});

//Qs. Take a number n as input from user. Create an array of numbers from 1 to n.
//Use the reduce method to calculate sum of all numbers in the array.
//Use the reduce method to calculate product of all numbers in the array.
n = prompt("Enter a number");
```

```
arr=[]
for(let i=1;i<=n;i++){
    arr[i-1]=i;
}
console.log(arr);

const sumResult = arr.reduce((prev,curr)=>{
    return prev+curr;
})
console.log(sumResult);

const productResult = arr.reduce((prev,curr)=>{
    return prev*curr;
})
console.log(productResult);
```

#DOM(Document object Model)

- Way to access HTML code by JS.
- When a web page is loaded, the browser cretes a DOM of the page
- Jo html ham likhte hain js ke sath attach krke vo sara html code js ke andrr access krskte hain kyuki automatically sara html js ke andrr akar object mai convert hojate h aur uss special object ko ham nam dete h document.
- This document is available in window object using which we can access our full HTML code.
- console.dir(window) is used to print object i.e here window object.
- So you can access or write html in js using dom for eg.. document.body.style.background="blue".

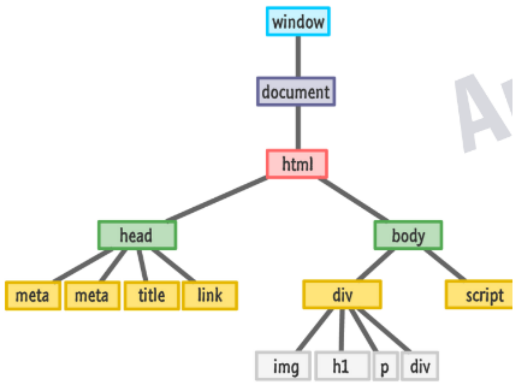
#The 3 Musketeers of Web Dev

- Html:- used to give structure to your website.
- Css:- Styling
- JS:- Logic or functionality

"<style> tag connects HTML with CSS and <script> tag   connects HTML with JS"

#Window Object

The window object represents an open window in a browser. It is browser's object(not Javascript) and is automatically created by Browser. It is a global object with lots of properties and methods.



```
console.log(window.alert("hello"))
```

#DOM Manipulation

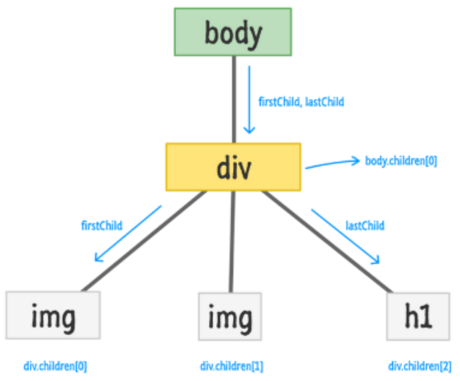
```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <link rel="stylesheet" href="styles.css" />
  </head>
  <body>
    <h1 id="majorhead">hy</h1>
    <p style="color: white" class="sectionpara">Section paragraph</p>
    <script src="script.js"></script>
  </body>
</html>
```

```
// 1. The getElementById method in JavaScript returns a single Element object
let myhead = document.getElementById("majorhead");
myhead.style.color = "white";
console.dir(myhead);

// 2. getElementsByClassName returns collection of elements(html collection)
console.log(sectionparag);
let sectionparag = document.getElementsByClassName("sectionpara");

//3. getElementByTagName returns Html Collection
let paratag = document.getElementsByTagName("p");
console.log(paratag);

//4. returns first elements
let parasection = document.querySelector(".sectionpara");
parasection.style.color = "brown";

//5. returns a NodeList
let Allparasection = document.querySelectorAll(".sectionpara");
console.log(Allparasection);
```

→Nodes in DOM TREE

In a Document Object Model (DOM) tree, there are three main types of nodes: text nodes, comment nodes, and element nodes. Here's an explanation of each:

1. **Element Nodes:**
    - Element nodes represent the HTML or XML tags in the document. For example, `<div>`, `<p>`, `<a>`, etc., are all represented as element nodes in the DOM tree.
2. **Text Nodes:**
    - Text nodes represent the textual content within an element.
    - For example, in the HTML `<p>Hello, world!</p>`, the text node contains the text "Hello, world!".
3. **Comment Nodes:**
    - Comment nodes represent HTML or XML comments.
    - For example, `<!-- This is a comment -->` would be represented as a comment node in the DOM tree.



```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <link rel="stylesheet" href="styles.css" />
  </head>
  <body>
  <h1 style="visibility: none">My First Heading</h1>
    <div class="main">
      <p>main div</p>
      <ul>
        <li>Mango</li>
        <li>Banana</li>
      </ul>
    </div>
    <script src="script.js"></script>
  </body>
</html>
```

```javascript
let maindiv = document.querySelector(".main");
console.log(maindiv.firstChild);
console.log(maindiv.lastChild);

console.log(maindiv.children[0]);
console.log(maindiv.children[1]);
```

→Important properties of DOM

- tagName : returns tag for element nodes
- innerText : returns the text content of the element and all its childeren
- innerHTML : returns the plain text or HTML contents in the elements
- textContent: returns textual content even for hidden elements.

```
let maindiv = document.querySelector(".main");
console.log(maindiv.innerText);
console.log(maindiv.innerHTML);

maindiv.innerText = "hy";
maindiv.innerHTML = "<p> gud morning </p>";

console.log(maindiv.innerText);
console.log(maindiv.innerHTML);
let headingfirst = document.querySelector("h1");
console.log(headingfirst.textContent);
```

Qs. Create a H2 heading element with text - “Hello JavaScript”. Append “from Apna College students” to this text using JS.

Qs. Create 3 divs with common class name - “box”. Access them & add some unique text to eachof them.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Adding Text to Divs with JavaScript</title>
</head>
<body>

<div class="box"></div>
<div class="box"></div>
<div class="box"></div>

<script>
    // Get all elements with the class name "box"
    var boxes = document.getElementsByClassName("box");

    // Add unique text to each div
    boxes[0].textContent = "Text for the first box";
    boxes[1].textContent = "Text for the second box";
    boxes[2].textContent = "Text for the third box";
</script>

</body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Appending Text with JavaScript</title>
</head>
<body>

<h2 id="heading">Hello JavaScript</h2>

<script>
    // Get the h2 element
    var headingElement = document.getElementById("heading");

    // Append text to the existing text content
    headingElement.textContent += " from Apna College students";
</script>

</body>
</html>
```

→More about DOM Manipulation

1. getAttribute(attr)
2. setAttribute(attr,value)
3. node.style
4. document.createElement(”el”):- to create elements
5. node.append(el) :- adds at the end of node(inside)
6. node.prepend(el):- adds at the start of node(inside)
7. node.before(el):- adds before the node(outside)
8. node.after(el) :- adds after the node(outiside)
9. node.remove() :- removes the node

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <link rel="stylesheet" href="styles.css" />
  </head>
  <body>
    <div class="box">first div</div>
    <p class="para">first para</p>

    <script>
      let box = document.querySelector(".box");
      let paragra = document.querySelector(".para");

      // Create a new paragraph element
      let para2 = document.createElement("p");

      // Set the text content of the paragraph
      para2.textContent = "second para";

      //add para2 inside and at the end of para
```

jstej

```
      paragra.append("para2");
    </script>
  </body>
</html>
```

#Events in JS

- The change in the state of an object is known as Event.
- Event are fired to notify code of “interesting changes” that may affect code execution.
- There are various events such as Mouse events(click, double click..) , Keyboard events , Form events , Print event and many more

→Event handling using inline handlers

```
<button
      ondblclick="console.log('clicked two times')"
      onclick="console.log('hello');alert('gud day')"
    >
      click me
    </button>

    <div onmouseover="alert('you are inside div')">this is my div</div>
```

⟶ Event handling using function(more priority than inline handlers)

```
let btnevent = document.querySelector(".bt1");
btnevent.onclick = () => {
  console.log("button clicked handler 1");
};
btnevent.onclick = () => {
  console.log("button clicked handler 2");
};
```

“in the above code the output when button is clicked will be button clicked handler 2”

⟶ Event handling using EventListeners.

```
let btnevent = document.querySelector(".bt1");
btnevent.addEventListener("click",() =>{
     console.log("button was clicked handler one");
});
btnevent.addEventListener("click",() =>{
     console.log("button was clicked handler 2");
});
const handler3 = ()=>{
  console.log("button was clicked handler 3");
};

btnevent.addEventListener("click",handler3);
btnevent.removeEventListener("click",handler3);
```

→Event object

- It is a special object that has details about the event

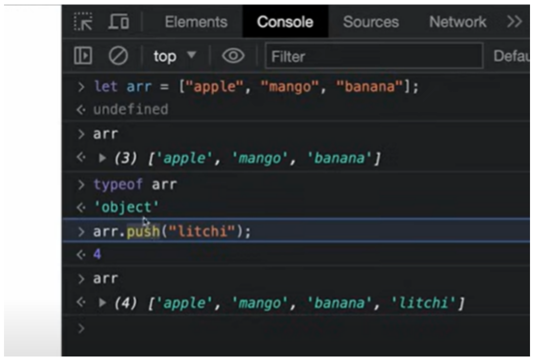
- All event handlers have access to the Event Objects properties and methods.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Event Object Example</title>
</head>
<body>
    <button id="myButton">Click me</button>
    <script>
        // Event handler function
        function handleClick(event) {
            // Accessing properties of the Event object
            console.log("Event type:", event.type);
            console.log("Target element:", event.target);
            console.log("X coordinate of mouse pointer:", event.clientX);
            console.log("Y coordinate of mouse pointer:", event.clientY);
        }

        // Get the button element
        var button = document.getElementById("myButton");

        // Add event listener to the button
        button.addEventListener("click", handleClick);
    </script>
</body>
</html>
```

#Classes and Objects

→Prototype in JS

- A javascript object is an entity having state and behavior (properties and method)
- JS objects have special property called prototype.
- we can set prototype using **proto**
- if object and prototype have same method , objects method will be used.
- for example ....Just like think you have created an array which is an object in js . Now when you try to find out console.log(typeof arr) you will get object as an output. when you click on that object you will find various properties and methods inside that object and also you will find an object named prototype which contains implementation of various array methods such as push , pop etc..
- prototype in short is basically reference to an object.



```
> let arr = ["apple", "mango", "banana"];
< undefined
> arr
< ▸ (3) ['apple', 'mango', 'banana']
> typeof arr
< 'object'
> arr.push("litchi");
< 4
> arr
< ▸ (4) ['apple', 'mango', 'banana', 'litchi']
>
```

```
//creating objects
const student = {
   fullName: "tejendra pal",
   marks: 94,
   printMarks: function (){
     console.log("marks = ", this.marks);
   },
};


//creating your own prototype
const employee = {
    calcTax() {
       console.log("tax rate is 10%");
    },
};

const karanArjun = {
      salary: 50000,
};

//the object karaArjun will also have a method of emplyee in its prototype
karanArjun.__proto__ = employee
karanArjun.calcTax();
//if both employee and karanArjun have same method then method of karanArjun will be executed
```

→Classes , objects and Constructor

- class is a program code template for creating objects.
- those objects will have some state(variables) and some behaviour (functions) inside it.
- classes are basically blueprint for creating objects.class is a single template that is used to create many objects.
- objects are basically refers to real world entity or an instance of class.
- Constructor() method is automatically invoked by new keyword while creating objects . It is basically used to initialize object

```
class ToyotaCar {
   constructor(brand,mileage){
      console.log("creating new object");
      this.brand =brand;
      this.mileage = mileage;//this is the reference to current instance of class
   }
   start(){
      console.log("start");
   }
   stop(){
     console.log("stop");
   }
 }

 let fortuner = new ToyotaCar("fortuner",10); //creating object
 let lexus = new ToyotaCar("lexus",12);
```

→Super keyword

- The super keyboard is used to call the constructor of its parent class to access parents property.
- If we have one parent class and one child class containing constructor in both class then we need to use super keyword in order to execute the code without any error.

→Inheritance

- Inheritance is passing down properties and methods from parent class to child class.

→Error Handling

- try-catch is used to handle errors if any error occurs inside try catch then code inside catch block gets executed without stopping code execution and throwing errors.

```
class Person {
   constructor(name){
      this.species = "home sapiens";
      this.name = name;
   }

   eat(){
     console.log("eat");
   }
}

class Engineer extends Person {
    constructor(name){
      super(name);//invoke parent class constructor
    }
    work() {
       try {
           super.eat();//invoke parent class method
           console.log("solve problems , build something");
       } catch (error) {
           console.error("Error occurred while working:", error);
       }
    }
}

let engObj = new Engineer("tej");
engObj.work();

 //output
/* eat
solve problems , build something */
```

#CALLBACK, PROMISES , ASYNC-AWAIT

→Synchronous

- Synchronous means the code runs in a particular sequence of instructions given in the program.
- Each instructions wait for the previous instruction to complete its execution.

→Asynchronous

- Due to synchronous programming , sometimes important instructions get blocked due to some previous instructions , which causes a delay in the UI.

- Asynchronous code execution allows to execute next instructions immediately and doesn't block the flow.

```
//Asynchronous programming
console.log("one")
console.log("two");

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

console.log("three");
console.log("four");

//Output
one
two
three
four
hello
```

→ Callback Function(function containing another function as an argument)

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

function calculator(a,b,sumCallback){
   sumCallback(a,b);
}

calculator(1,2,(a,b)=>{
  console.log(a+b);
});
```

→ Consider Below code:-

```
function getData(dataId){
    setTimeout(()=>{
      console.log("data",dataId);
    },2000);
}
getData(1);
getData(2);
getData(3);

//output (all output will be printed together simultaneously)
data 1
data 2
data 3
```

- What to do if you want to print above output separately one by one i.e first print "data 1" then after 2 seconds "data 2" then after 2 seconds print "data 2". This can be done using callback hell. for example …you have created an app in which user has to first enter there first name and if the first name is valid then only password is checked.
- Callback Hell : Nested callbacks stacked below one another forming a pyramid structure(pyramid of doom). Due to which the style of programming becomes difficult to understand and manage.

```
function getData(dataId, getNextData){
 setTimeout(() =>{
    console.log("data" , dataId);
    if(getNextData){
       getNextData();
    }
  },2000);
}

//callback
getData(1,()=>{
    console.log("getting data2....");
    getData(2,() =>{
         console.log("getting data3....");
         getData(3,()=>{
           console.log("getting data4...");
           getData(4);
         });
       });
    });
```

→Promise

- Promise is for "eventual" completion of task. it is an object in JS. It is a solution to callback hell.
- resolve and reject are callbacks provided by JS.

```
let promise = new Promise((resolve, reject)=>{
  console.log("I am a promise");
  resolve(123);
});
```

- So a promise can have three state pending, fulfilled and rejected.
  - Pending:-   the result is undefined
  - Resolved:- the result is a value
  - Rejected:- the result is an error object

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

  let promise = getPromise();
  promise.then((res) => {
     console.log("promise fullfilled",res);
  });

  promise.catch((err)=>{
     console.log("rejected",err);
 });
```

- Promise Chaining:-
  - Reason why promise chaining is used in JavaScript is **Sequential Execution**: Promise chaining allows you to execute asynchronous operations in a specific sequence, ensuring that one operation completes before the next one starts.

```
//problem
function asyncFunc1(){
    return new Promise((resolve,reject)=>{
      setTimeout(()=>{
        console.log("data1");
        resolve("success");
      },4000);
    });
}

function asyncFunc2(){
    return new Promise((resolve,reject)=>{
      setTimeout(()=>{
        console.log("data2");
        resolve("success");
      },4000);
    });
}

console.log("fetching data1...");
let p1 =  asyncFunc1();
p1.then((res) =>{
   console.log(res);
});

console.log("fetching data2...");
let p2 =  asyncFunc2();
p2.then((res) =>{
   console.log(res);
});

//output (both the promise will get executed at the same time)
fetching data1...
fetching data2...
data1
success
data2
success
```

```

//solution
//to execute promises one by one
function asyncFunc1() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log("data1");
            resolve("success");
        }, 4000);
    });
}

function asyncFunc2() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log("data2");
            resolve("success");
        }, 4000);
    });
}

//promise chaining
console.log("fetching data1...");
asyncFunc1()
```

jstej

```
    .then((res) => {
        console.log(res);
        console.log("fetching data2...");
        return asyncFunc2();
    })
    .then((res) => {
        console.log(res);
    });

//output
fetching data1...
data1
success
fetching data2...
data2
success
```

→ Async and Await

- Async function always returns a promise
- Await pauses the execution of its surroundings async function until the promise is settled.

Async and await in JavaScript solve several common issues related to asynchronous programming:

1. **Callback Hell**: Async/await provides a more structured and readable way to handle asynchronous operations compared to nested callbacks, which can lead to deeply nested and hard-to-read code.
2. **Sequential Code**: Async/await allows you to write asynchronous code that looks synchronous. This makes it easier to write and understand sequential code flows, where one operation depends on the result of another.
3. **Promise Chain**: Async/await simplifies working with Promise chains. Instead of chaining `.then()` calls, you can use `await` to pause execution until a Promise is resolved or rejected, making the code more linear and easier to follow.

```
function getData(dataId){
   return new Promise((resolve,reject)=>{
    setTimeout(()=>{
      console.log("data",dataId);
    },2000);
  });
}
async function getAllData(){
await getData(1);
await getData(2);
await getData(3);
}
```

→IIFE: Immediately Invoked Function Expression

- IIFE is a function that is called immediately as soon as it is defined
- An IIFE is like a bubble that keeps everything inside it, so it doesn't mess with anything outside. It's handy for organizing code, keeping things private, and avoiding conflicts with other scripts. You wrap your code in a function and then immediately call it, so it runs right away.

```
function getData(dataId){
   return new Promise((resolve,reject)=>{
    setTimeout(()=>{
      console.log("data",dataId);
    },2000);
  });
}
(async function getAllData(){
  console.log("getting data1....");
```

jstej

```
    await getData(1);
    console.log("getting data2....");
    await getData(2);
    console.log("getting data3....");
    await getData(3);
})();
```

#Fetch API

- The Fetch API provides an interface for fetching (sending and receiving) resources.
- It uses Request and Response objects.
- The fetch() method is used to fetch a resource(data).
- Syntax :- let promise = fetch(url,[options]). if options not given by default the get request us considered

```
const URL ="https://cat-fact.herokuapp.com/facts";

const getFacts = async ()=>{
  console.log("getting data.....")
  let response = await fetch(URL);
  console.log(response);//JSON-FORMAT
  let data = await response.json();
  console.log(data[0].text);
};
```

→Understanding Terms

- AJAX is Asynchronous JS and XML.
- JSON is JavaScript Object Notation.
- json() method : returns a second promise that resolves with the result of parsing the response body text as JSON.(Input is JSON,output is JS object)

```
//using promises
function getFacts() {
   fetch(URL)
     .then((response) => {
        return response.json();
     })
     .then((data) => {
         console.log(data);
         factPara.innerText = data[2].text;
     };
 }
```

→HTTP AND STATUS CODES

Hypertext Transfer Protocol (HTTP) is an application-layer protocol for transmitting hypermedia documents, such as HTML. It was designed for communication between web browsers and web servers, but it can also be used for other purposes. HTTP follows a classical client-server model, with a client opening a connection to make a request, then waiting until it receives a response

HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped in five classes:

1. Informational responses (`100` – `199`)
2. Successful responses (`200` – `299`)

3. Redirection messages (`300` – `399`)
4. Client error responses (`400` – `499`)
5. Server error responses (`500` – `599`)