JS

Async js is only performed in

1. setInterval
2. setTimeout
3. promises
4. axios
5. fetch
6. XMLHttpRequest

JS is Single threaded ont multithreaded

In JavaScript, the `async` function allows you to write asynchronous code in a way that looks synchronous, making it easier to read and write. An `async` function automatically returns a promise, and within an `async` function, you can use the `await` keyword to pause the execution of the function until a promise is resolved.

Here's a basic example of how `async` functions work:

```
javascriptCopy code
// An example async function
async function fetchData() {
  try {
    // Pauses here until the promise is resolved
    let response = await fetch('https://jsonplaceholder.typic
ode.com/posts');
    let data = await response.json(); // Pauses until respons
e.json() is resolved

    console.log(data);
  } catch (error) {
    console.log('Error:', error);
  }
```

```
}

// Calling the async function
fetchData();
```

Key Points:

1. `async` **Keyword**: When you declare a function as `async`, it automatically returns a promise. If the function returns a value, that value is wrapped in a resolved promise. If it throws an error, the promise is rejected.
2. `await` **Keyword**: The `await` keyword can only be used inside an `async` function. It waits for the promise to resolve and returns the result. If the promise is rejected, it throws the error.
3. **Error Handling**: You can use `try...catch` blocks inside `async` functions to handle errors that might occur during the execution of the promises.

Example with Error Handling:

```
javascriptCopy code
async function getUserData() {
  try {
    let response = await fetch('https://api.example.com/user/
1');
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    let userData = await response.json();
    return userData;
  } catch (error) {
    console.error('Error fetching user data:', error);
  }
}

getUserData()
```

```
  .then(data => console.log(data))
  .catch(error => console.error('Error in getUserData:', erro
r));
```

Summary:

- Use `async` to declare an asynchronous function.
- Use `await` to wait for a promise to resolve before proceeding with the function's execution.
- Handle errors with `try...catch` blocks within the `async` function.

Promise in JavaScript

A **Promise** in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It allows you to write asynchronous code in a more manageable and readable way.

Key States of a Promise:

1. **Pending**: Initial state, neither fulfilled nor rejected.
2. **Resolved**: The operation completed successfully, and the promise has a result.
3. **Rejected**: The operation failed, and the promise has a reason for the failure.

Creating a Promise:

You can create a promise using the `Promise` constructor, which takes a function with two parameters: `resolve` and `reject`.

Example:

```
javascriptCopy code
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
```

```
      const data = { name: "Alice" };
      resolve(data); // Simulate a successful API call
    }, 2000);
  });
}

fetchData()
  .then((data) => {
    console.log("Data received:", data); // Logs: "Data recei
ved: { name: 'Alice' }"
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

Summary:

- **Promises** manage asynchronous operations.
- They start in a **pending** state, can be **fulfilled** with `resolve`, or **rejected** with `reject`.
- Use `.then()` to handle success and `.catch()` for errors.

Async and Await

Basically we use Promise to write async code but in promises we use .then for the result of the promise so hence to avoid .then we use async and await

> jab bhi koi code async hai tab hume wait krna pdta hai kyuki hume nhi pata uska answer kab aayega then we use await

To rewrite the code using `async` and `await`, you can directly await the promise instead of using `.then()`. Here's how the code would look:

```
javascriptCopy code
const p = new Promise((resolve, reject) => {
  resolve("Promise resolved value");
});

async function getData() {
  const res = await p;
  console.log(res);
}

getData();
```

Explanation:

- **Promise**: The promise `p` resolves with the value `"Promise resolved value"`.
- `async` **Function**: The `getData` function is declared as `async`, which allows you to use the `await` keyword inside it.
- `await`: The `await p` pauses the execution of `getData` until the promise `p` resolves, and assigns the resolved value to `res`.
- `console.log(res)`: After the promise is resolved, the value is logged to the console.

Here's a small code snippet that uses `fetch` to retrieve data from an API and then uses `await` to convert the response to JSON:

```
javascriptCopy code
async function fetchData() {
  const response = await fetch('https://jsonplaceholder.typic
ode.com/posts/1');
  const data = await response.json();
```

```
  console.log(data);
}

fetchData();
```

Explanation:

- `fetch`: The `fetch` function is used to make a network request to the provided URL.
- `await fetch`: The `await` keyword is used to wait for the `fetch` request to complete and return a response.
- `response.json()`: Once the response is received, `await` is used again to wait for the response to be converted to JSON.
- `console.log(data)`: Finally, the JSON data is logged to the console.

Here's the code snippet with `try...catch` added for error handling:

```
javascriptCopy code
async function fetchData() {
  try {
    const response = await fetch('https://jsonplaceholder.typ
icode.com/posts/1');

    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}
`);
    }

    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error fetching data:', error);
  }
```

```
}

fetchData();
```

Explanation:

- `try...catch`: The `try` block is used to wrap the code that might throw an error. If any error occurs during the fetch or JSON conversion process, it will be caught in the `catch` block.
- **Error Handling**: If the `fetch` request fails (e.g., due to network issues) or the

```
javascriptCopy code
async function fetchData() {
  try {
    const response = await fetch('https://jsonplaceholder.typ
icode.com/posts/1');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error fetching data:', error);
  }
}

fetchData();
```

Explanation:

- `try...catch`: The `try` block attempts to fetch the data and convert it to JSON. If anything goes wrong (like network issues), the `catch` block handles the error and logs it to the console.

Concurrency

In JS whenever Async and sync code runs simultaenously the it is called as concurrency

Parallelism

Parallelism in JavaScript refers to the ability to perform multiple tasks simultaneously, despite JavaScript being single-threaded by nature.

- **Web Workers**: Run code in parallel threads, ideal for heavy tasks.
- **Promises &** `async/await`: Manage asynchronous tasks concurrently.
- **Service Workers**: Handle background tasks independently of web pages.

Throttling

Throttling is a technique used to limit the number of times a function is called over a period of time. It ensures that a function doesn't execute too frequently, even if triggered repeatedly.

In JavaScript, "undefined" and "not defined" are related but distinct concepts. Here's the difference between them:

Undefined

- **Definition:** `undefined` is a value that indicates that a variable has been declared but has not been assigned a value.
- **Example:**

  ```
  let a;
  console.log(a); // Output: undefined
  ```

  In this example, the variable `a` is declared but not assigned a value, so when you try to access it, it returns `undefined`.

- **Another Case:** A function can also return `undefined` if it doesn't have a return statement.

```
function foo() {}
console.log(foo()); // Output: undefined
```

Not Defined

- **Definition:** "Not defined" occurs when you try to access a variable that has not been declared at all. This results in a `ReferenceError`.
- **Example:**
  In this case,
  `b` has never been declared in the code, so trying to access it throws a `ReferenceError`.

```
console.log(b); // ReferenceError: b is not defined
```

Summary

- **`undefined`:** A variable has been declared, but no value has been assigned to it.
- **`not defined`:** A variable has not been declared at all, and trying to access it results in a `ReferenceError`.

The difference lies in **how** `let` and `const` behave after being hoisted compared to `var`. Although `let` and `const` are hoisted, they are not initialized during the hoisting process, which creates a **Temporal Dead Zone (TDZ)** where accessing them before their declaration will throw an error.

Event Loop: Main and Side Stack (Short Version)

1. **Main Stack**: Executes synchronous code.
   - `console.log("Start")` → `asyncOperation()` → `console.log("End")`.

2. **Side Stack (Callback Queue)**: Holds asynchronous callbacks until the main stack is clear.
    - `setTimeout` schedules its callback after 1000 ms.
3. **Event Loop**: Moves callbacks from the side stack to the main stack for execution.

Example Execution:

1. Logs `"Start"`.
2. Schedules `setTimeout` callback.
3. Logs `"End"`.
4. After 1000 ms, logs `"Async operation"` and `"Callback executed"`.

Output:

```
Start
End
Async operation
Callback executed
```

Key Differences Between `var`, `let`, and `const` Regarding Hoisting:

1. `var` Hoisting:

- **Declaration and Initialization:**
    - `var` variables are both hoisted and initialized with `undefined`.
    - This means you can access them before their declaration, but they will return `undefined`.
- **Example:**

```
console.log(a); // Output: undefined
var a = 5;
```

```
console.log(a); // Output: 5
```

Here, `a` is hoisted and initialized with `undefined` before the code executes, so no error is thrown when `console.log(a)` is called before `a` is declared.

2. `let` and `const` Hoisting:

- **Declaration but No Initialization:**
  - `let` and `const` variables are hoisted, but they are not initialized. They enter the TDZ until the line where they are declared is reached.
  - Trying to access them before their declaration results in a `ReferenceError`.
- **Example with `let`:**

```
console.log(b); // ReferenceError: Cannot access 'b' befor
e initialization
let b = 10;
console.log(b); // Output: 10
```

- **Example with `const`:**

```
console.log(c); // ReferenceError: Cannot access 'c' befor
e initialization
const c = 20;
console.log(c); // Output: 20
```

In these examples, `b` and `c` are hoisted but not initialized, which means they remain in the TDZ until the actual declaration is encountered in the code. If you try to access them before that point, it results in a `ReferenceError`.

Why It Matters:

- **Safety and Predictability:**
  - The TDZ ensures that `let` and `const` variables are not accessed before they are properly initialized, reducing potential bugs and making the code more predictable.

- **Different Behavior:**
  - `var` allows access to the variable before it's actually declared (though it may return `undefined`), which can lead to unexpected behavior.
  - `let` and `const` enforce stricter rules by not allowing access until the point of declaration, leading to more robust and clear code.

Summary:

- **`var`:** Hoisted and initialized with `undefined`, accessible before declaration.
- **`let` and `const`:** Hoisted but not initialized, leading to the TDZ, and not accessible before declaration.

1. Primitive vs. Reference Types

- **Primitive Types**: These include `Number`, `String`, `Boolean`, `Null`, `Undefined`, `Symbol`, and `BigInt`. These are stored directly in the variable and are immutable. When you assign or pass a primitive value, a copy of that value is made.

```
javascriptCopy code
let a = 10;
let b = a; // b gets a copy of a
b = 20;
console.log(a); // Outputs: 10 (a is unchanged)
```

- **Reference Types**: These include `Object`, `Array`, `Function`, etc. Variables that hold reference types store the reference (or memory address) to the actual object in memory, not the object itself. When you assign or pass an object, you're copying the reference, not the object.

```
javascriptCopy code
let obj1 = { name: "Alice" };
```

```
let obj2 = obj1; // obj2 references the same object as obj
1
obj2.name = "Bob";
console.log(obj1.name); // Outputs: Bob (obj1 is affected
because obj1 and obj2 reference the same object)
```

2. Mutating Reference Types

Since reference types store a reference to the object, changing the properties of an object via one variable will reflect in all variables that reference that object.

```
javascriptCopy code
let arr1 = [1, 2, 3];
let arr2 = arr1;

arr2.push(4);
console.log(arr1); // Outputs: [1, 2, 3, 4] (arr1 is affected
because arr1 and arr2 reference the same array)
```

3. Passing Reference Types to Functions

When you pass an object or array to a function, you're passing the reference to that object, meaning the function can modify the original object.

```
javascriptCopy code
function modifyArray(arr) {
    arr.push(5);
}

let numbers = [1, 2, 3];
modifyArray(numbers);
console.log(numbers); // Outputs: [1, 2, 3, 5] (numbers is mo
```

```
dified inside the function)
```

4. Cloning Reference Types

To avoid unintended side effects, you can clone an object or array to create a new copy that doesn't share the same reference.

- **Shallow Copy** (using `Object.assign` or the spread operator):

```
javascriptCopy code
let obj1 = { name: "Alice" };
let obj2 = { ...obj1 }; // shallow copy
obj2.name = "Bob";
console.log(obj1.name); // Outputs: Alice (obj1 is not aff
ected)
```

- **Deep Copy** (using `JSON.parse(JSON.stringify(...))` or libraries like Lodash):

```
javascriptCopy code
let obj1 = { name: "Alice", address: { city: "Wonderland"
} };
let obj2 = JSON.parse(JSON.stringify(obj1)); // deep copy
obj2.address.city = "Atlantis";
console.log(obj1.address.city); // Outputs: Wonderland (ob
j1 is not affected)
```

Shadowing in JavaScript occurs when a variable declared within a certain scope (like a block, function, or inner scope) has the same name as a variable in an outer scope. The inner variable "shadows" or overrides the outer variable within its scope.

Shadowing with `var`, `let`, and `const`:

1. `var`:

- `var` is function-scoped, so shadowing with `var` typically occurs within functions or nested functions.
- If `var` is declared in a block, it doesn't create block-level scope, so it can lead to unexpected shadowing.

```
var x = 10;
function foo() {
    var x = 20; // Shadows the outer 'x'
    console.log(x); // 20
}
foo();
console.log(x); // 10 (the outer 'x' remains unchanged)
```

2. `let`:

- `let` is block-scoped, meaning it creates a new scope within blocks `{}`.
- Shadowing with `let` occurs when a `let` variable is declared in an inner block with the same name as an outer variable.

```
let y = 10;
{
    let y = 20; // Shadows the outer 'y' only within this
block
    console.log(y); // 20
}
console.log(y); // 10 (the outer 'y' remains unchanged)
```

3. `const`:

- `const` behaves similarly to `let` in terms of scope and shadowing.
- A `const` variable can shadow another variable with the same name in an outer scope.

```
const z = 10;
{
    const z = 20; // Shadows the outer 'z' only within thi
s block
    console.log(z); // 20
}
console.log(z); // 10 (the outer 'z' remains unchanged)
```

Summary:

- `var`: Function-scoped, can be shadowed within functions but doesn't create block-level scope, leading to potential confusion in shadowing.
- `let` and `const`: Block-scoped, can shadow variables in outer scopes within a block, providing more predictable and safer scoping behavior.

Closures in JavaScript

Closures are a fundamental concept in JavaScript that allow functions to "remember" the environment in which they were created, even after that environment has finished executing. In simpler terms, a closure is a function that retains access to its lexical scope, even when the function is executed outside that scope.

Key Points:

1. **Lexical Scope:**
    - JavaScript has lexical (or static) scoping, meaning that the scope of variables is determined by their location within the source code.
    - Inner functions have access to variables declared in their outer functions.
2. **How Closures Work:**
    - When a function is defined inside another function, it has access to variables of the outer function.

- Even after the outer function has executed and returned, the inner function (closure) retains access to the outer function's variables.

3. **Practical Example:**

```
function outerFunction() {
    let outerVariable = 'I am from the outer scope';

    function innerFunction() {
        console.log(outerVariable); // Accesses the outer
function's variable
    }

    return innerFunction;
}

const closure = outerFunction(); // outerFunction executes
and returns innerFunction
closure(); // "I am from the outer scope" is logged
```

- **Explanation:**
  - `innerFunction` is a closure. It has access to `outerVariable` even after `outerFunction` has finished executing.
  - When `closure()` is called, it still remembers the value of `outerVariable` from the scope in which it was created.

4. **Common Use Cases:**

- **Data Privacy:** Closures are often used to create private variables or functions that are not accessible from outside the scope.
- **Function Factories:** Closures can be used to create functions with preset arguments.
- **Event Handlers:** Closures are useful in setting up event handlers that maintain access to variables defined in a different scope.

```
function createCounter() {
    let count = 0;
    return function() {
        count += 1;
        return count;
    };
}

const counter = createCounter();
console.log(counter()); // 1
console.log(counter()); // 2
console.log(counter()); // 3
```

- **Explanation:** The returned function is a closure that has access to the `count` variable. Every time `counter()` is called, it increments `count` and returns the new value.

5. **Closures and Loops:**
    - When dealing with closures inside loops, it's important to understand how closures capture variables.
    - If not handled carefully, closures inside loops can lead to unexpected behavior due to all closures capturing the same variable (which might change over iterations).

```
for (var i = 1; i <= 3; i++) {
    setTimeout(function() {
        console.log(i); // Prints 4, 4, 4 because the clos
ure captures the final value of i
    }, 1000);
}
```

- **Fix Using `let`:** Since `let` is block-scoped, each iteration creates a new `i`, avoiding the issue.

```
for (let i = 1; i <= 3; i++) {
    setTimeout(function() {
        console.log(i); // Prints 1, 2, 3 as expected
    }, 1000);
}
```

Summary:

- A closure is a function that retains access to its outer scope's variables even after the outer scope has finished executing.
- Closures are powerful for creating private variables, maintaining state, and ensuring functions have access to the right variables when they're executed later on.
- Understanding closures is crucial for writing effective JavaScript code, particularly in asynchronous operations, callbacks, and event handlers.

In JavaScript, `unshift` and `shift` are methods used to manipulate arrays.

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

```
javascriptCopy code
let arr = [2, 3, 4];
arr.unshift(1);  // arr becomes [1, 2, 3, 4]
```

- `shift()`: Removes the **first** element from an array and returns that element. This method changes the length of the array.

```
javascriptCopy code
let arr = [1, 2, 3, 4];
let firstElement = arr.shift();  // arr becomes [2, 3, 4],
firstElement is 1
```

`splice()` modifies an array by removing, adding, or replacing elements:

- **Remove:** `arr.splice(start, deleteCount)`
- **Add:** `arr.splice(start, 0, item1, item2, ...)`
- **Replace:** `arr.splice(start, deleteCount, item1, item2, ...)`

Example:

```
javascriptCopy code
let arr = [1, 2, 3];
arr.splice(1, 1, 'a');  // Replaces 2 with 'a' -> [1, 'a', 3]
```

Window in js

Js has many features but there are so many features which are not there so js uses the features which are not the part of the js but a part of the browser features and js use it is called as window

var adds itself to the window

let doesnt add it

```
alert("hellooooo");

promt("Enter the name");
```

Browser Context API

In JavaScript, particularly in the context of browser automation tools like Puppeteer or Playwright, the **Browser Context API** allows you to create and manage multiple isolated browsing sessions within a single browser instance.

Key Points:

- **Isolation**: Each browser context operates independently, with its own cookies, storage, cache, and session data. This means actions in one context do not affect others.
- **Incognito Mode**: Browser contexts can be used to simulate incognito or private browsing sessions.
- **Efficiency**: Multiple contexts can run simultaneously in one browser instance, making it efficient for testing or multi-user scenarios.

Example (Puppeteer):

```
javascriptCopy code
const puppeteer = require('puppeteer');

(async () => {
  const browser = await puppeteer.launch();
  const context = await browser.createIncognitoBrowserContext
(); // Create new context
  const page = await context.newPage(); // Open new page in t
hat context
  await page.goto('https://example.com');
  // Interact with the page in an isolated session
  await browser.close();
})();
```

In this example, a new browser context is created, allowing actions on `example.com` without sharing data like cookies or local storage with other contexts.

Heap Memory

Whatever variables we made that data should be stored somewhere and that is called as heap memory

execution context

It is a container that contains the function code and it is executed whenever the function is called and contain three things, 1.variable 2.function 3.lexical environment

Lexical Environment

It is like a chart that defines the particular function access which type of data which not, that means it hold scope and scope chains ex. the parent function cannot access its nested function data but child can access its parent variables etc.

Truthy and Falsy Values in JavaScript

- **Truthy**: Values that evaluate to `true` in a boolean context (e.g., non-zero numbers, non-empty strings).
- **Falsy**: Values that evaluate to `false` in a boolean context (e.g., `0`, `""`, `null`, `undefined`, `false`, `NaN`).

Example:

```
javascriptCopy code
if ("hello") console.log("Truthy");  // Output: Truthy
if (0) console.log("Falsy");         // No output, because 0
is falsy
```

Call back Function (async js)

A call back function is a normal function that runs after completion of async function

A code that runs afterwards we give a function to that code and after completion then run that particular function, that function is just like the normal function and that is called as call back

`setTimeout` in JavaScript

The `setTimeout` function is used to execute a piece of code after a specified delay. It takes two main parameters:

1. **Callback Function**: The function to execute after the delay.
2. **Delay**: The time, in milliseconds, to wait before executing the callback function.
3. setTimeout(callback, delay);

```
console.log("Start");

setTimeout(() => {
  console.log("Executed after 2 seconds");
}, 2000);

console.log("End");
```

First class functions

It is a function that is stored in a variable and we can use it by the var name directly

- **Assignment**: Functions can be assigned to variables.

```
javascriptCopy code
const greet = function(name) {
  return `Hello, ${name}!`;
```

```
};
console.log(greet("Alice")); // Output: Hello, Alice!
```

- **Passing as Arguments**: Functions can be passed as arguments to other functions.

```
javascriptCopy code
function processUser(callback) {
  const user = "Bob";
  callback(user);
}

processUser(function(name) {
  console.log(`User: ${name}`);
}); // Output: User: Bob
```