Class Notes

Synchronous

- Synchronous means each task at a time, JS will execute a single task then it will move to the next task
- It is also called blocking code and JS engine has to wait for the remaining lines of code to execute
- In this code will execute top to bottom

Asynchronous

- Async means task will be executed at last after all the sync statements, once the callstack is empty. Event loop will only push the callback or promises only when call stack is empty
- Microtask queue has higher priority than callback queue
- Code will wait in Web API until it time expires
- Then the callback function will be moved to the callback queue then after event loop will constantly check the callstack to empty
- When callstack will become empty then that callback function will moved to the callstack then that executes
- Microtask queue have more priority than callback queue

JavaScript Engine

- JS engine is part of browser
- In the browser only **callstack** is said to be V8 engine all other are part of browser like web API, event loop etc

Promise

- It is a object
- It represents eventual completion of an async task

Node.js Synchronous vs Asynchronous Programming Notes

Synchronous Programming

Key Characteristics:

- **One task at a time**: JavaScript executes a single task, then moves to the next task
- **Blocking code**: The JS engine waits for each line to complete before moving forward
- **Top-to-bottom execution**: Code runs sequentially from top to bottom
- **Predictable flow**: Each statement must finish before the next one starts

Example:

```
console.log("start");
for (let i = 0; i <= 10; i++) {
    console.log(i);
}
console.log("middle");
console.log("end");
```

Asynchronous Programming

Key Characteristics:

- **Non-blocking**: Tasks execute without waiting for previous tasks to complete
- **Event Loop dependent**: Async tasks execute only when the call stack is empty
- **Queue system**: Uses callback queue and microtask queue for task management
- **Web API integration**: Code waits in Web API until conditions are met

Execution Flow:

1. Async code is moved to **Web API**
2. Code waits until timer expires or condition is met
3. Callback function moves to **Callback Queue**
4. **Event Loop** constantly monitors the call stack
5. When call stack is empty, callback moves from queue to call stack
6. Function executes

Priority System:

- **Microtask Queue** > **Callback Queue**
- Microtask queue has higher priority and executes first

JavaScript Engine Architecture

Browser Components:

- **Call Stack**: Part of V8 engine
- **Web API**: Browser feature (not part of JS engine)
- **Event Loop**: Browser feature
- **Callback Queue**: Browser feature
- **Microtask Queue**: Browser feature

> ***Important***: *Only the call stack is part of the V8 engine; everything else is provided by the browser.*

Promises

Definition:

- **Object** that represents the eventual completion of an asynchronous task
- Has three states:
  - **Pending**: Initial state
  - **Fulfilled**: Operation completed successfully
  - **Rejected**: Operation failed

Code Examples

Example 1: Basic Asynchronous Behavior

```
setTimeout(() => {
    console.log("Inside Timeout");
}, 0);

console.log("Start");

for (let i = 0; i < 2000; i++) {
    console.log(i);
}
```

Key Points:

- `0` represents **minimum time to wait**
- **Maximum time** depends on code length and execution time
- Even with 0ms delay, setTimeout is still asynchronous

Example 2: Multiple Timeouts

```
console.log(1);

setTimeout(() => {
    console.log(2);
}, 2000)

setTimeout(() => {
    console.log(3);
}, 1000)

console.log(4);
```

Execution Order: 1 → 4 → 3 (after 1s) → 2 (after 2s)

Example 3: Event Loop in Action

```
setTimeout(() => {
    console.log("timeout 1");
});

for (let i = 0; i < 2000; i++) {
    console.log("");
}

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

Behavior: Synchronous code (loop + console.log) executes first, then timeout callback

Promise Example: Fetch API

```
let promise = fetch("https://jsonplaceholder.typicode.com/posts");

promise
    .then((value) => {
        console.log(value);
        let jsonData = value.json();
        console.log(jsonData);

        jsonData
            .then((data) => {
                console.log(data);
            }).catch((err) => {
                console.log(err);
            });
        console.log("inside then");
    })
    .catch((err) => {
        console.log(err);
        console.log("Inside catch");
    });
```

Key Takeaways

1. **Synchronous** = Blocking, sequential execution
2. **Asynchronous** = Non-blocking, uses event loop and queues
3. **Event Loop** only processes queues when call stack is empty
4. **Microtask Queue** has higher priority than Callback Queue
5. **Promises** represent future values and use microtask queue
6. **setTimeout(fn, 0)** still goes through async flow, not immediate execution
7. **Browser architecture** separates JS engine (V8) from Web APIs and Event Loop