3. JavaScript Asynchronous

1. Key Characteristics of JavaScript

- ***Synchronous***
  - Code executes top-down - one line at a time!

```
const firstFunction = () => {
  console.log("First Function");
};

const secondFunction = () => {
  console.log("Second Function");
};

firstFunction();
secondFunction();

// First Function
// Second Function
```

- ***Blocking***
  - Subsequent processes won't kick off until the former is completed - no matter how long the first one takes.
- ***Single-Threaded***
  - Only one thread exists - *main thread* - used for executing any code.
- Key Problems
  - It might be possible that *fetchDataFromDB* could take substantial time.
  - The code cannot proceed forward without executing the first line.

- If it does, JavaScript will throw an error.

```
let response = fetchDataFromDB('endpoint')
displayDataFromDB(response)
```

---

2. Need for Async JavaScript

- Vanilla JavaScript - Not enough.
- Web Browsers - allows defining functions that should not be executed synchronously and should be invoked when some event occurs.
- Functions can be executed on various prompts
  - passage of time - set Timeout or set Interval
  - interaction with mouse - add Event Listener
  - arrival of data - callbacks, promises, async-await

---

3. Timeouts and Interval

- Traditional methods of running code in an async manner
  - after a set time period - set Timeout.

```
//Without Parameter

const greet = () => {
  console.log("Hey There!");
};

setTimeout(greet, 2000);

//With Parameter
const greet = (name) => {
  console.log(`Hey ${name}`);
};

setTimeout(greet, 2000, "Nikhil");

//Clear Timeout

const executeFun = setTimeout(greet, 2000, "Nikhil");
clearTimeout(executeFun);

// Nothing will be printed on the console - if we clear the function using timeout.
// Clear Timeout - Can be used when we unmount the piece of code to free resources.
```

  - after regular intervals - set Interval.

```
const greet = (name) => {
  console.log(`Hey ${name}`);
};

const myGreet = setInterval(greet, 2000, "Nik");
clearInterval(myGreet);
```

- **Noteworthy Points**
  - Timers and Intervals - Not inbuilt features of JavaScript.
  - These are implemented by the browser.
  - Specified delay is the minimum delay and not the guaranteed delay.
  - We can achieve - the same effect of set Interval by using recursive Set Timeout.

```
setTimeout(
  (myFun = () => {
    console.log("Hello");
    setTimeout(myFun, 1000);
  }),
  1000
);

// Executes the function after every second - used setTimeout recursively.
```

---

4. Callbacks

- Functions - First Class Objects
  - *Callbacks* are functions that are passed as argument to other functions.
  - A function can be returned as a value from another function - *Higher Order Function.*
- **Synchronous Callbacks**
  - Callbacks that are executed immediately.
- **Asynchronous Callbacks**
  - Callbacks that are used to continue or resume code after execution of the async function.
  - These allow for delaying the execution of a function.
  - Example - Data Fetching
    - Such a process takes time and we need to delay the functions that need this particular data.
  - set Timeout and event listeners are key examples of async callback.

- If there are multiple levels of callbacks function - nesting can be troublesome and the overall code could be highly difficult to read and maintain - *callback hell.*

---

5. Promises

- Promise → Async Operation → Return Value
  - Promise Fulfilled → Success Callback
  - Promise Rejected → Failure Callback
- A promise is a proxy for a value.
- This value is not necessarily known when the promise is created.
- Allows to associate handlers with async action's success or failure.
- Promise → Simply an object
  - Key States
    - *Pending*
      - Initial State → Neither fulfilled nor rejected.
    - *Fulfilled*
      - Operation completed successfully.
    - Rejected
      - Operation failed.
- Promises → facilitate dealing with async codes in simpler way → avoid occurrence of callback hell.

Analogy	JavaScript
1. Your friend	1. Promise
2. Can Get Tacos / Can Not Get Tacos	2. Promise Value
3. Can Get Tacos	3. Promise Fulfilled
4. Can Not Get Tacos	4. Reject Promise
5. Set Up Table	5. Success Callback
6. Cook Pasta	6. Failure Callback

---

7. Promises - Part II

- We can pass on Rejection as a second parameter to then function.
- However, it is not the encouraged approach.

```
then() function

const promise = New Promise((resolve, reject) => {
resolve() or reject()
})

promise.then(onFulfillment)
promise.catch(onRejection)

or

promise.then(onFulfillment, onRejection)
```

- We can also perform chaining in promises, if required.

```
promise.then(fn).catch(fn)
```

- This solves the problem of callback hell by allowing a more concise syntax.

Static Methods

- Promise All
  - This can be used when we have to query multiple APIs before finally rendering the data.
  - The method takes promises and traverse through them which finally returns an array of results.
  - If any of the promises reject, this would reject all promises with the first error message.
- Promise All Settled
  - Same as Promise All - but this returns even if one of the promise fails.
  - This only focuses on completing the promises irrespective of the output state.
- Promise Race
  - This returns a promise that fulfills or rejects - as soon as one input promise fulfils or rejects.
  - If we provide two promises - A and B and the second one fulfils first - it would return with the second promise.

---

8. Async and Await

- Introduced in ES8 (ES2017)

Async

- Async - Await: Allows us to write sync-looking code while performing async task bts.

- Async functions are instances of Async Function constructor.
- Unlike normal functions, async allows us to return a promise.

```
// Normal Function
function greet(){
  console.log('Hello')
}

greet();

// Hello

//Async Function
async function greet() {
  return "Hello";
}

console.log(greet());

// Promise { 'Hello' }

// To print the data

async function greet() {
  return "Hello";
}

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

// Hello
```

Await

- await can be used with async promise function to pause the code until promise if settled and result is returned
- Cannot be used with normal function

```
async function greet() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("Hello");
    }, 1000);
  });

  let result = await promise;
  console.log(result);
}

greet();

// Here the await keywords pauses the further execution of program until promise value is returned.
```

- Sequential vs Concurrent vs Parallel Execution
  - Seq - First A → then B

```
// Sequential Exeuction

function resolveHello() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Hello");
    }, 2000);
  });
}

function resolveWorld() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("World");
    }, 1000);
  });
}

async function sequential() {
  const hello = await resolveHello();
  console.log(hello);

  const world = await resolveWorld();
  console.log(world);
}

sequential();

// Exeuction Time : 3 Seconds
```

- Con - A and B

```
async function sequential() {
  const hello = await resolveHello();
  const world = await resolveWorld();

  console.log(hello);
  console.log(world);
}

sequential();

// Execution Time : 2 Seconds
```

- Parallel - Whichever executes first.
  - We can use Promise All for this.
  - Pass in an array of promises.