

## Asynchronous JavaScript

- Asynchronous JavaScript refers to the programming paradigm in which tasks (operations or functions) are executed without waiting for the previous task to finish.
- In asynchronous programming, certain operations, such as data fetching or timers, can be executed in the background, allowing the main thread (JavaScript's single-threaded event loop) to continue running without being blocked.
- Asynchronous JavaScript is essential for modern web applications to ensure they remain responsive and efficient while performing tasks like API calls, timers, and file operations.

### Key Concepts in Asynchronous JavaScript

- **Synchronous vs. Asynchronous Execution**

**Synchronous:** Tasks are performed one after another, and each task waits for the previous one to complete.

**Asynchronous:** Tasks are started, and JavaScript can continue executing other code while waiting for those tasks to complete (without blocking).

- **Event Loop:** JavaScript uses an event loop to handle asynchronous operations. When an asynchronous task is complete (e.g., data is fetched from a server), the event loop adds a callback function to the task queue, which is executed after the synchronous code finishes.

### Mechanisms for Asynchronous JavaScript

- Callbacks (already discussed)
- Promises
- Async/Await

## Promise

- A **Promise** in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
- It's a cleaner way to manage asynchronous operations compared to callbacks, improving readability and maintainability of code.
- Promises are one of the most exciting additions to JavaScript ES6.
- For asynchronous programming, earlier versions of JavaScript used callbacks, among other things.
- Promises in JavaScript are object representations of asynchronous operations.
- A promise starts in a pending state and ends in either a fulfilled (resolved) state or a rejected state.
- Promises are a pattern that greatly simplifies asynchronous programming by making the code look synchronous and avoiding problems associated with callbacks.
- Promises are generally used with the Fetch API to fetch data from a JSON file from remote server or a specific location.
- A promise has two possible outcomes: it will either be kept when the time comes, or it won't.

### Syntax of a Promise

```
let promise = new Promise(function(resolve, reject) {  
    // asynchronous operation  
    let success = true; // this simulates the outcome  
  
    if (success) {  
        resolve("Operation succeeded!");  
    } else {  
        reject("Operation failed.");  
    }  
});
```

Where,

- **resolve(value)**: When the operation is successful, the resolve function is called.
- **reject(error)**: When the operation fails, the reject function is called.

## How Promises Work

Promises have three states:

- **Pending:** Initial state, neither fulfilled nor rejected.
- **Fulfilled:** The operation was successful, and `resolve()` get called.
- **Rejected:** The operation failed, and `reject()` get called.

In order to create a promise in JavaScript, we use the Promise constructor.

```
new Promise(); // Promise Constructor
```

The Promise constructor accepts a function as an argument. This function is called the executor. The executor is called automatically when the Promise constructor is invoked.

```
let promise = new Promise(function() {  
  // Executor function  
});
```

The executor accepts two callback functions, conventionally named `resolve()` and `reject()`.

```
let promise = new Promise(function(resolve, reject) {  
  // Executor function with "resolve" & "reject"  
});
```

Inside the executor, we manually call the `resolve()` function if the operation completes successfully, and invoke the `reject()` function in case an error occurs.

The status of a promise is fulfilled if we invoke the `resolve` method, and the status of the promise is rejected if we invoke the `reject` method.

## Difference Between Callbacks and Promises

Callbacks and Promises both handle asynchronous operations, but they work differently.

- **Callbacks:** Functions passed as arguments to be executed once an asynchronous operation completes. They can lead to callback hell if nested deeply.

### Example

```
function fetchData(callback) {  
  setTimeout(() => {  
    callback("Data received!");  
  }, 1000);  
}  
fetchData((data) => {  
  console.log(data); // "Data received!"  
});
```

- **Promises:** Provide a more structured way of handling asynchronous operations, chaining .then() methods to avoid deep nesting (callback hell).

### Example

```
let promise = new Promise((resolve) => {  
  setTimeout(() => {  
    resolve("Data received!");  
  }, 1000);  
});  
promise.then((data) => console.log(data));
```

## Key Differences

- **Readability:** Promises improve readability by avoiding deeply nested code (callback hell).
- **Error Handling:** Promises use .catch() for error handling, whereas callbacks require error handling at each level.
- **Chaining:** Promises allow method chaining (.then()), making it easier to manage sequential asynchronous tasks.

## Async Promises Requests

### Example

```
let completed = true;
let learnJS = new Promise(function (resolve, reject) {
  setTimeout(() => {
    if (completed) {
      resolve("I have completed learning JS.");
    } else {
      reject("I haven't completed learning JS yet.");
    }
  }, 3 * 1000);
});
```

### Example - Promise Resolved

```
let promise = Promise.resolve("Resolved")
console.log(promise)
```

### Example - Promise Rejected

```
let promise = Promise.reject("Rejected")
console.log(promise)
```

### Example - Promises Resolved

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

### Example - Promises Rejected

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

### Example

```
let completed = true;
let learnJS = new Promise(function(resolve, reject) {
  setTimeout(() => {
    if (completed) {
      resolve("I have completed learning JS.");
    } else {
      reject("I haven't completed learning JS yet.");
    }
  }, 3 * 1000);
});
console.log(learnJS);
```

### Example

```
var isMomHappy = true;
var willGetNewPhone = new Promise(
  function(resolve, reject) {
    if (isMomHappy) {
      var phone = {
        brand: 'Apple',
        color: 'black',
        model: '14 pro max',
        storage: '256 gb'
      };
      resolve(phone); // fulfilled
    } else {
      var reason = new Error('mom is not happy');
      reject(reason); // reject
    }
  }
);
console.log(willGetNewPhone);
```

## With "error()" Object

It is recommended to pass an Error object since it helps in debugging by viewing the stacktrace.

### Example

```
let promise = new Promise(function(resolve, reject) {  
    setTimeout(() => reject  
    (new Error("Whoops!")), 1000);  
});
```

### Example

```
let promise = new Promise(function(resolve, reject) {  
    resolve("done");  
    reject(new Error("Error"));  
    setTimeout(() => resolve("Promise Resolved"));  
});  
console.log(promise);
```

### Example

```
let promise = new Promise((resolve, reject) => {  
    if (Math.random() * 100 <= 90) {  
        resolve('Hello, Promises!');  
    }  
    reject(new Error('In 10% of the cases, I fail. Miserably.'));  
});  
console.log(promise);
```

## Consumer Functions (Promise Instance Methods)

Consumer functions, or Promise instance methods, allow us to handle the results of promises. The most common promise instance methods are:

- `.then()`
- `.catch()`
- `.finally()`

These methods help manage the asynchronous flow of data, providing ways to access, manipulate, and handle promise states.

1. **.then():** The `.then()` method is used to handle a fulfilled promise and execute a callback function once the promise resolves. It takes two optional arguments:
  - A callback for the resolved (fulfilled) state.
  - A callback for the rejected state (although `.catch()` is preferred for error handling).

Syntax

```
promise.then(onFulfilled, onRejected);
```

Example

```
let promise = new Promise((resolve, reject) => {  
  setTimeout(() => resolve("Data received!"), 1000);  
});  
promise.then((result) => {  
  console.log(result); // Output: "Data received!"  
});
```

Use Case

- When you want to perform an action after a promise is resolved, such as handling API responses or user inputs.
- **Chaining with .then():** You can chain multiple `.then()` calls to sequentially handle multiple asynchronous operations.

Example

```
let promise = new Promise((resolve) => resolve(10));  
promise  
  .then((num) => num * 2)    // Doubles the number  
  .then((num) => num + 5)    // Adds 5 to the result  
  .then((num) => console.log(num)); // Output: 25
```



2. **.catch():** The .catch() method is used to handle errors or rejections in a promise chain. It catches any errors or rejections that occur in the promise or in any of the .then() handlers.

**Syntax**

```
promise.catch(onRejected);
```

**Example**

```
let promise = new Promise((resolve, reject) => {
    setTimeout(() => reject("Error occurred!"), 1000);
});
promise
    .then((result) => console.log(result))
    .catch((error) => console.error(error)); // Output: "Error occurred!"
```

**Use Case**

- Error handling in API requests or any operation that could fail asynchronously.

3. **.finally():** The .finally() method is called when a promise is settled, regardless of whether it was resolved or rejected. It allows you to perform cleanup tasks or final actions after the promise has been handled.

**Syntax**

```
promise.finally(onFinally);
```

**Example**

```
let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("Task completed!"), 1000);
});
promise
    .then((result) => console.log(result)) // Output: "Task completed!"
    .catch((error) => console.error(error))
    .finally(() => console.log("Operation finished")); // Output: "Operation
                                                    finished"
```

**Use Case**

- Cleanup tasks, such as stopping a loading spinner or closing a file handle, regardless of whether the promise succeeded or failed.

## Promise chaining

- Promise chaining is a powerful concept in JavaScript that allows you to handle multiple asynchronous operations sequentially.
- Each `.then()` in the chain returns a new promise, enabling further operations on the result of the previous promise.

### Example: Basic Promise Chaining

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve(5), 1000); // Resolves with value 5 after 1 second
});

promise
  .then((result) => {
    console.log(result); // Output: 5
    return result * 2; // Pass 10 to the next `.then()`
  })
  .then((result) => {
    console.log(result); // Output: 10
    return result + 5; // Pass 15 to the next `.then()`
  })
  .then((result) => {
    console.log(result); // Output: 15
  });
```

### Example: Error Handling in Promise Chains

```
function execute() {
  return new Promise((resolve, reject) => {
    setTimeout(() => reject("Error Occured!"), 1000);
  });
}

execute()
  .then((result) => {
    console.log(result);
    return result + " processed"; // This won't execute due to rejection
  })
```


```
.catch((error) => {
  console.error("Error:", error); // Output: "Error Occured!"
})
.then(() => {
  console.log("This will execute regardless of the previous error."); // Always
                                                                    executes
});
```

### Example: Chaining with Multiple Promises

```
function firstTask() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("First task completed"), 1000);
  });
}
function secondTask() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Second task completed"), 500);
  });
}
function thirdTask() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Third task completed"), 200);
  });
}
firstTask()
  .then((result) => {
    console.log(result); // Output: "First task completed"
    return secondTask(); // Chain second task
  })
  .then((result) => {
    console.log(result); // Output: "Second task completed"
    return thirdTask(); // Chain third task
  })
  .then((result) => {
    console.log(result); // Output: "Third task completed"
  });
```

### Example: Returning Promises in a Chain

```
function execute(value) {  
    return new Promise((resolve) => {  
        setTimeout(() => resolve(value), 1000);  
    });  
}  
execute(10)  
    .then((value) => {  
        console.log(value); // Output: 10  
        return execute(value + 5); // Return new promise for value 15  
    })  
    .then((value) => {  
        console.log(value); // Output: 15  
        return execute(value + 15); // Return new promise for value 30  
    })  
    .then((value) => {  
        console.log(value); // Output: 30  
    });
```

The logo for Learn 2 Earn Labs is positioned behind the code. It features the word 'LEARN' in blue, a large red '2' inside a hexagon, the word 'EARN' in blue, and 'LABS' in grey below it.