Chapter 8

Asynchronous JavaScript

**Synchronous Code**

* **Example**

console.log('Start');

console.log('Processing...');

console.log('End');

**Callbacks**: Functions passed as arguments to other functions to execute later.

console.log('Start');

setTimeout(() => console.log('Processing...'), 2000);

console.log('End');

**Promises**: Objects representing the eventual completion or failure of an asynchronous operation.

console.log('Start');

const promise = new Promise((resolve) => {

setTimeout(() => resolve('Processing...'), 2000);

});

promise.then(console.log);

console.log('End');

**Async/Await**: Simplifies asynchronous code by allowing it to be written as if it were synchronous.

async function fetchData() {

console.log('Start');

const result = await new Promise((resolve) =>

setTimeout(() => resolve('Processing...'), 2000)

);

console.log(result);

console.log('End');

}

fetchData();

**Challenges of Asynchronous Code**

1. **Callback Hell**: Nested callbacks become hard to manage and read:

fetchData((data) => {

processData(data, (result) => {

saveData(result, (success) => {

console.log('Done');

});

});

});

**Example: Using a Callback**

function fetchData(callback) {

setTimeout(() => {

console.log('Data fetched');

callback('Data');

}, 1000);

}

function processData(data) {

console.log('Processing:', data);

}

fetchData(processData);

**Example of Callback Hell**

fetchData((data) => {

processData(data, (processedData) => {

saveData(processedData, (response) => {

console.log('Data saved:', response);

});

});

});

**Named Functions**: Break down nested callbacks into named functions to improve readability.

function fetchDataCallback(data) {

processData(data, processDataCallback);

}

function processDataCallback(processedData) {

saveData(processedData, saveDataCallback);

}

function saveDataCallback(response) {

console.log('Data saved:', response);

}

fetchData(fetchDataCallback);

**Promises**: Promises provide a better way to handle asynchronous operations by chaining .then() calls, avoiding deep nesting.

fetchData()

.then(processData)

.then(saveData)

.then((response) => console.log('Data saved:', response))

.catch((error) => console.error('Error:', error));

**Async/Await**: The async/await syntax simplifies asynchronous code, making it look synchronous.

async function handleData() {

try {

const data = await fetchData();

const processedData = await processData(data);

const response = await saveData(processedData);

console.log('Data saved:', response);

} catch (error) {

console.error('Error:', error);

}

}

handleData();

**Example of Creating a Promise**

const fetchData = () => {

return new Promise((resolve, reject) => {

setTimeout(() => {

const success = true; // Simulating success or failure

if (success) {

resolve('Data fetched successfully');

} else {

reject('Error fetching data');

}

}, 2000);

});

};

**Example of Consuming a Promise**

fetchData()

.then((message) => {

console.log(message);

})

.catch((error) => {

console.error(error);

})

.finally(() => {

console.log('Operation completed');

});

**Example of Promise Chaining**

const processData = (data) => {

return new Promise((resolve) => {

setTimeout(() => {

resolve(`${data} processed`);

}, 2000);

});

};

const saveData = (data) => {

return new Promise((resolve) => {

setTimeout(() => {

resolve(`${data} saved`);

}, 2000);

});

};

fetchData()

.then((data) => processData(data))

.then((processedData) => saveData(processedData))

.then((result) => {

console.log(result); // Output: Data fetched successfully processed saved

})

.catch((error) => {

console.error('Error:', error);

});

#### **Error Handling in Promises**

#### **Example**

fetchData()

.then((data) => {

throw new Error('Processing error');

})

.then((processedData) => saveData(processedData))

.catch((error) => {

console.error('Caught an error:', error.message);

});

**Promise Combinators**

**Promise.all():** Resolves when all promises in an array resolve. Rejects if any promise is rejected.

Promise.all([fetchData(), processData('test'), saveData('data')])

.then((results) => console.log('All results:', results))

.catch((error) => console.error('Error:', error));

**Promise.race():** Resolves or rejects as soon as the first promise settles.

Promise.race([fetchData(), processData('test')])

.then((result) => console.log('First settled:', result))

.catch((error) => console.error('Error:', error));

**Promise.allSettled():** Returns an array of all promise outcomes, regardless of whether they resolve or reject.

Promise.allSettled([fetchData(), processData('test')])

.then((results) => console.log('Results:', results));

**Promise.any():** Resolves with the first fulfilled promise, ignoring rejections.

Promise.any([fetchData(), processData('test')])

.then((result) => console.log('First fulfilled:', result))

.catch((error) => console.error('All rejected:', error));

**Promise Chain with .catch()**

fetch('https://api.example.com/data')

.then((response) => {

if (!response.ok) {

throw new Error('Network response was not ok');

}

return response.json();

})

.then((data) => {

console.log('Data:', data);

})

.catch((error) => {

console.error('Error:', error.message);

});

**Example of Error Propagation**

const fetchData = () => {

return new Promise((resolve, reject) => {

setTimeout(() => {

reject('Failed to fetch data');

}, 2000);

});

};

fetchData()

.then((data) => {

console.log('Data:', data);

return 'Next operation';

})

.then((next) => {

console.log(next);

})

.catch((error) => {

console.error('Error caught:', error);

});

### **Using** .catch() **for Multiple Errors**

#### **Example**

fetch('https://api.example.com/data')

.then((response) => {

if (!response.ok) {

throw new Error('Response not OK');

}

return response.json();

})

.then((data) => {

console.log('Data:', data.someNonExistentProperty); // This will throw an error

})

.catch((error) => {

console.error('Caught error:', error.message);

});

**Single .catch() for a Chain**: Place one .catch() at the end of a promise chain to handle all errors.

fetch('https://api.example.com/data')

.then((response) => response.json())

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

.catch((error) => console.error('Error:', error));

**Logging and Debugging**: Log meaningful error messages to debug issues effectively.

.catch((error) => console.error('An error occurred:', error.message));

**Graceful Recovery**: Provide fallback behavior when an error occurs.

.catch((error) => {

console.error('Error:', error.message);

return 'Fallback data';

});

**Handling Errors Without .catch()**

**Example**

const fetchData = async () => {

try {

const response = await fetch('https://api.example.com/data');

if (!response.ok) {

throw new Error('Failed to fetch data');

}

const data = await response.json();

console.log('Data:', data);

} catch (error) {

console.error('Error caught:', error.message);

}

};

fetchData();

**Example: Using async**

async function fetchData() {

return 'Data fetched!';

}

fetchData().then((result) => console.log(result)); // Output: Data fetched!

**Example: Using await**

async function fetchData() {

const response = await fetch('https://api.example.com/data');

const data = await response.json();

console.log('Fetched data:', data);

}

fetchData();

**Example: Sequential Execution**

async function processSequential() {

console.log('Fetching user...');

const user = await fetch('https://api.example.com/user').then((res) => res.json());

console.log('User fetched:', user);

console.log('Fetching posts...');

const posts = await fetch(`https://api.example.com/posts?userId=${user.id}`).then((res) => res.json());

console.log('Posts fetched:', posts);

return { user, posts };

}

processSequential();

**Example: Handling Errors**

async function fetchData() {

try {

const response = await fetch('https://api.example.com/data');

if (!response.ok) {

throw new Error('Network response was not ok');

}

const data = await response.json();

console.log('Data:', data);

} catch (error) {

console.error('Error fetching data:', error.message);

}

}

fetchData();

**Sequential Execution**

Promises are awaited one after another.

async function fetchSequentially() {

const user = await fetch('https://api.example.com/user').then((res) => res.json());

const posts = await fetch(`https://api.example.com/posts?userId=${user.id}`).then((res) => res.json());

console.log('User:', user, 'Posts:', posts);

}

fetchSequentially();

**Concurrent Execution**

async function fetchConcurrently() {

const [user, posts] = await Promise.all([

fetch('https://api.example.com/user').then((res) => res.json()),

fetch('https://api.example.com/posts').then((res) => res.json())

]);

console.log('User:', user, 'Posts:', posts);

}

fetchConcurrently();

### **Returning Values from** async **Functions**

#### **Example**

async function add(a, b) {

return a + b;

}

add(5, 10).then((result) => console.log(result)); // Output: 15

**Example: Using async**

async function fetchData() {

return 'Data fetched!';

}

**Example: Using await**

async function fetchData() {

const response = await fetch('https://api.example.com/data');

const data = await response.json();

console.log('Fetched data:', data);

}

fetchData();

**Example: Sequential Execution**

async function processSequential() {

console.log('Fetching user...');

const user = await fetch('https://api.example.com/user').then((res) => res.json());

console.log('User fetched:', user);

console.log('Fetching posts...');

const posts = await fetch(`https://api.example.com/posts?userId=${user.id}`).then((res) => res.json());

console.log('Posts fetched:', posts);

return { user, posts };

}

processSequential();

**Example: Handling Errors**

async function fetchData() {

try {

const response = await fetch('https://api.example.com/data');

if (!response.ok) {

throw new Error('Network response was not ok');

}

const data = await response.json();

console.log('Data:', data);

} catch (error) {

console.error('Error fetching data:', error.message);

}

}

fetchData();

**Sequential Execution**

Promises are awaited one after another.

async function fetchSequentially() {

const user = await fetch('https://api.example.com/user').then((res) => res.json());

const posts = await fetch(`https://api.example.com/posts?userId=${user.id}`).then((res) => res.json());

console.log('User:', user, 'Posts:', posts);

}

fetchSequentially();

**Concurrent Execution**

async function fetchConcurrently() {

const [user, posts] = await Promise.all([

fetch('https://api.example.com/user').then((res) => res.json()),

fetch('https://api.example.com/posts').then((res) => res.json())

]);

console.log('User:', user, 'Posts:', posts);

}

fetchConcurrently();

**Returning Values from async Functions**

**Example**

async function add(a, b) {

return a + b;

}

add(5, 10).then((result) => console.log(result)); // Output: 15

* **Cannot Use Outside async Functions**:

The await keyword requires an async function wrapper.

// This will throw an error

const data = await fetch('https://api.example.com/data');

**fetch()**

**Basic Syntax**

fetch(url, options)

.then((response) => {

// Handle response

})

.catch((error) => {

// Handle error

});

**Example: Basic Fetch Request**

fetch('https://api.example.com/data')

.then((response) => {

if (!response.ok) {

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

}

return response.json(); // Parse JSON response

})

.then((data) => console.log(data)) // Process data

.catch((error) => console.error('Fetch error:', error));

**Example: Fetching JSON Data**

async function fetchData() {

try {

const response = await fetch('https://api.example.com/users');

if (!response.ok) {

throw new Error(`Failed to fetch: ${response.statusText}`);

}

const data = await response.json(); // Parse response as JSON

console.log('User Data:', data);

} catch (error) {

console.error('Error:', error);

}

}

fetchData();

**Example: Sending Data with POST**

async function postData(url = '', data = {}) {

try {

const response = await fetch(url, {

method: 'POST',

headers: {

'Content-Type': 'application/json',

},

body: JSON.stringify(data), // Convert object to JSON string

});

if (!response.ok) {

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

}

const result = await response.json();

console.log('Server Response:', result);

} catch (error) {

console.error('Error:', error);

}

}

// Example usage

postData('https://api.example.com/create', { name: 'John', age: 30 });

**Example: Handling Errors Gracefully**

fetch('https://api.example.com/data')

.then((response) => {

if (!response.ok) {

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

}

return response.json();

})

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

.catch((error) => {

console.error('An error occurred:', error.message);

});

**Example: Adding Custom Headers**

fetch('https://api.example.com/secure-data', {

method: 'GET',

headers: {

'Authorization': 'Bearer token123',

'Accept': 'application/json',

},

})

.then((response) => response.json())

.then((data) => console.log('Secure Data:', data))

.catch((error) => console.error('Error:', error));

**Example: Fetching Related Data**

fetch('https://api.example.com/user/1')

.then((response) => response.json())

.then((user) => {

console.log('User:', user);

return fetch(`https://api.example.com/posts?userId=${user.id}`);

})

.then((response) => response.json())

.then((posts) => console.log('User Posts:', posts))

.catch((error) => console.error('Error:', error));

**Example: Concurrent Fetching**

async function fetchMultiple() {

try {

const [user, posts] = await Promise.all([

fetch('https://api.example.com/user/1').then((res) => res.json()),

fetch('https://api.example.com/posts').then((res) => res.json()),

]);

console.log('User:', user);

console.log('Posts:', posts);

} catch (error) {

console.error('Error:', error);

}

}

fetchMultiple();

**Example: Timeout with AbortController**

const controller = new AbortController();

const timeout = setTimeout(() => controller.abort(), 5000); // Abort after 5 seconds

fetch('https://api.example.com/data', { signal: controller.signal })

.then((response) => response.json())

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

.catch((error) => console.error('Fetch aborted or failed:', error))

.finally(() => clearTimeout(timeout));

**setTimeout: Scheduling Code Execution After a Delay**

**Syntax**

const timeoutId = setTimeout(callback, delay, arg1, arg2, ...);

**Example: Delayed Message**

setTimeout(() => {

console.log('This message appears after 2 seconds');

}, 2000);

Using clearTimeout

You can cancel a setTimeout before it executes using clearTimeout.

Example: Cancelling a Timer

const timerId = setTimeout(() => {

console.log('This will not run');

}, 5000);

clearTimeout(timerId); // Cancels the timer

**setInterval: Repeating Code Execution at Intervals**

**Syntax**

const intervalId = setInterval(callback, interval, arg1, arg2, ...);

**Example: Repeating Message**

setInterval(() => {

console.log('This message appears every 3 seconds');

}, 3000);

Using clearInterval

You can stop a repeating interval using clearInterval.

Example: Stopping an Interval

const intervalId = setInterval(() => {

console.log('Repeating message');

}, 1000);

setTimeout(() => {

clearInterval(intervalId); // Stops the interval after 5 seconds

}, 5000);

**Example: Passing Arguments**

setTimeout((name) => {

console.log(`Hello, ${name}!`);

}, 2000, 'Alice'); // Output: Hello, Alice!

**Example: Using setInterval with Arguments**

setInterval((message, count) => {

console.log(`${message} - Count: ${count}`);

}, 1000, 'Repeating', 5);

**Timer Use Cases**

* **Animation or Visual Effects**

let position = 0;

const moveBox = setInterval(() => {

position += 10;

document.getElementById('box').style.left = position + 'px';

if (position >= 300) clearInterval(moveBox); // Stops after moving 300px

}, 100);

* **Auto-Saving Data**

setInterval(() => {

console.log('Auto-saving user data...');

}, 5000);

* **Delaying Execution**

document.getElementById('button').addEventListener('click', () => {

setTimeout(() => {

console.log('Button clicked! This message is delayed.');

}, 1000);

});

**Example: Using Nested setTimeout**

function repeatAction() {

console.log('Repeated action');

setTimeout(repeatAction, 1000); // Repeats after 1 second

}

repeatAction();

**Timer Limitations and Considerations**

**Example**

setTimeout(() => console.log('Delayed'), 1000);

while (true) {} // Blocks execution

**Combining setTimeout and setInterval**

**Example: Countdown Timer**

let countdown = 10;

const timer = setInterval(() => {

console.log(`Time left: ${countdown}`);

countdown -= 1;

if (countdown < 0) {

clearInterval(timer);

console.log('Countdown finished!');

}

}, 1000);

**Example: Aggregating API Calls**

const fetchUser = fetch('/api/user'); // Fetch user data

const fetchPosts = fetch('/api/posts'); // Fetch posts data

const fetchComments = fetch('/api/comments'); // Fetch comments data

Promise.all([fetchUser, fetchPosts, fetchComments])

.then(([userResponse, postsResponse, commentsResponse]) => {

return Promise.all([

userResponse.json(),

postsResponse.json(),

commentsResponse.json(),

]);

})

.then(([user, posts, comments]) => {

console.log('User:', user);

console.log('Posts:', posts);

console.log('Comments:', comments);

})

.catch((error) => {

console.error('An error occurred:', error);

});

**Use Case: Batch Processing**

const numbers = [1, 2, 3, 4];

const squarePromises = numbers.map((num) =>

new Promise((resolve) => setTimeout(() => resolve(num \* num), 1000))

);

Promise.all(squarePromises).then((squares) => {

console.log('Squares:', squares); // Output: Squares: [1, 4, 9, 16]

});

**Example: Fastest Response**

const fastPromise = new Promise((resolve) => setTimeout(() => resolve('Fast!'), 500));

const slowPromise = new Promise((resolve) => setTimeout(() => resolve('Slow...'), 2000));

Promise.race([fastPromise, slowPromise]).then((result) => {

console.log('Winner:', result); // Output: Winner: Fast!

});

**Example: Timeout for a Network Request**

const networkRequest = fetch('/api/data');

const timeout = new Promise((\_, reject) =>

setTimeout(() => reject(new Error('Request timed out')), 3000)

);

Promise.race([networkRequest, timeout])

.then((response) => {

console.log('Network response:', response);

})

.catch((error) => {

console.error(error.message); // Output: Request timed out (if timeout wins)

});

**Combining Promise.all and Promise.race**

**Example: Resolving All Promises with a Timeout**

const fetchData = fetch('/api/data');

const timeout = new Promise((\_, reject) =>

setTimeout(() => reject(new Error('Request timed out')), 5000)

);

Promise.race([

Promise.all([fetchData, someOtherAsyncTask()]),

timeout

])

.then(([dataResponse, otherTaskResult]) => {

console.log('Data:', dataResponse);

console.log('Other Task:', otherTaskResult);

})

.catch((error) => {

console.error(error.message);

});