Programming JS

Margit Tennosaar

Summary

- Destructuring → Extract values from arrays/objects easily
- Spread & Rest Operators → Expand/collect values dynamically
- Method Chaining → Efficient, readable array operations
- Optional Chaining (?.) → Prevents errors with missing data

Asynchronous JavaScript

JavaScript is single-threaded, meaning it executes one task at a time.

Asynchronous operations prevent the browser from "freezing" while waiting for tasks like:

- Fetching data from an API
- Reading files
- Running timers (setTimeout)

Without async behaviour, web apps would be slow and unresponsive

What Is Asynchronous Code?

Synchronous (Blocking) → Code runs in order, one task at a time.

Asynchronous (Non-blocking) → Code continues executing while waiting for operations to finish.

```
console.log("Start");
setTimeout(() => console.log("Async Task Done"), 2000);
console.log("End");
```

```
Start
End
Async Task Done (after 2 seconds)
```

Callbacks

A function passed as an argument to another function, executed later.

```
function fetchData(callback) {
  setTimeout(() => callback("Data loaded"), 2000);
}
fetchData((data) => console.log(data)); // Expected output: "Data loaded" (after 2 seconds)
```

Too many nested callbacks make code hard to read & debug!

```
getData(function (a) {
  getMoreData(a, function (b) {
    getMoreData(b, function (c) {
      console.log(c);
    });
  });
});
```

Promises

A promise represents a value that might be available now, later, or never.

Three states:

- Pending → The async operation is in progress.
- Fulfilled → The operation was successful.
- Rejected → The operation failed.

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Success!"), 2000);
});
```

Handling promises

Promises make async operations readable & structured.

```
fetch("https://api.example.com/data")
  .then(response => response.json()) // Handle success
  .then(data => console.log(data))
  .catch(error => console.error("Error:", error)); // Handle failure
```

Async/Await

async/await makes asynchronous code look synchronous & easier to read.

- async marks a function as asynchronous.

- await pauses execution until the promise resolves.

```
async function fetchData() {
  try {
    const response = await fetch("https://api.example.com/data");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Error:", error);
  }
}
```

No need for .then(), just clean & readable code!

Promises vs Async/Await

Feature	Promises (.then()/.catch())	Async/Await
Readability	+ Can get complex when chaining	+ Looks like sync code
Error Handling	- Needs .catch() for every step	+ Uses try...catch
Nesting	- Can lead to chaining hell	+ Cleaner structure
Performance	+ Non-blocking, executes in background	+ Same as Promises

Which one to use?

Approach	Pros	Cons
Callback	Simple for small tasks	Hard to read, leads to Callback Hell
Promise	More readable, avoids nesting	Can get long with .then() chaining
Async/Await	Clean, readable, best for complex async tasks	Requires modern JavaScript ES8

Fetch API

JavaScript's fetch() is used to retrieve data from APIs.

```
fetch("https://jsonplaceholder.typicode.com/posts")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error("Error:", error));
```

Common API operations:

GET → Retrieve data (default).

POST → Send data.

PUT/PATCH → Update data.

DELETE → Remove data.

Fetch API with Async/Await

```
async function fetchData() {
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/posts");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Error:", error);
  }
}
fetchData();
```

Error handling

```
async function fetchDataWithErrorHandling() {
  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);
  } catch (error) {
    console.error("Fetch error:", error);
  }
}
```

Always check response.ok before processing data!

Using Real APIs for Practice

- JSONPlaceholder → Fake REST API for testing
  - https://jsonplaceholder.typicode.com/posts
- The Cat API → Get random cat pictures
  - https://api.thecatapi.com/v1/images/search
- OpenWeatherMap → Get weather data (API key required)
- https://api.openweathermap.org/data/2.5/weather?q=London&appid=your_api_key

Test these APIs using fetch() or async/await!

Summary

Asynchronous JavaScript lets code run while waiting for tasks to complete!

Callbacks → The old way, leads to Callback Hell.

Promises → Cleaner, avoids nesting, but .then() chaining can get long.

Async/Await → The modern approach, reads like synchronous code.

Fetch API → The standard for working with external APIs.