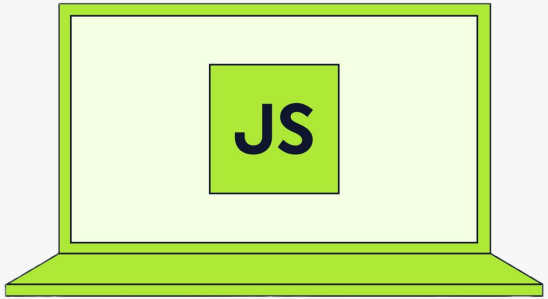
Newton School of Technology

The Complete Javascript Course



@newtonschool

Lecture 13: async / await

-Vishal Sharma

JS

Table of Contents

Newton School of Technology

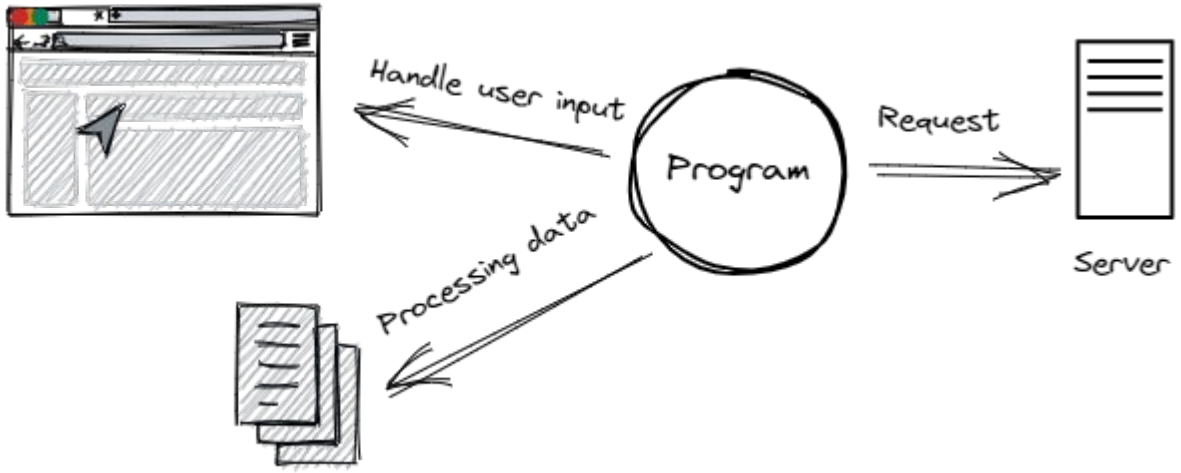
Newton School of Technology

Quick Revision

Promises and Callbacks

Pre-discussed: Async Programming

Asynchronous programming lets tasks run independently, allowing your program to continue other tasks while waiting for long operations to finish.

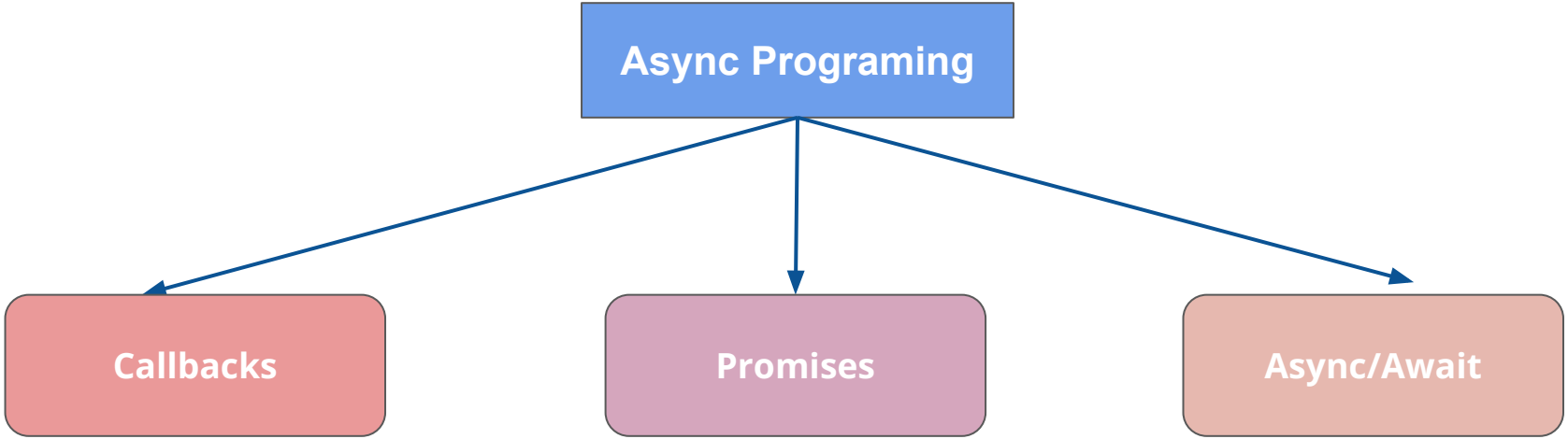


Here, the program sends a request to the server and, while waiting for the response, continues accepting and processing user inputs.

Pre-discussed: How to implement?

Newton School of Technology

There are three main ways to implement asynchronous programming in JavaScript:



Pre-discussed: Callbacks

Newton School of Technology

In simple terms, a callback is just a function that you give to another function. The receiving function will then decide when and how to execute.

```
function task1(callback) {
    console.log("Task 1 completed");
    callback();
    // Execute the callback after Task 1
}

function task2() {
    console.log("Task 2 completed");
}

task1(task2);
// Passing task2 as a callback to task1
```

Here it is upto task1() function when and where it executes callback function. Here it is executing it immediately but that is not always the case.

Pre-discussed: Callbacks

Newton School of Technology

Let’s add a delay of 2000 milliseconds before callback executes using setTimeout().

```javascript
function task1(callback) {
    setTimeout(() => {
      console.log("Task 1 completed");
      callback();
      // Execute the callback once Task 1 is done
    }, 2000); // Simulate a delay
}

function task2() {
    console.log("Task 2 completed");
}

task1(task2);
// Passing task2 as a callback to task1
```

Here we are adding a delay inside task1 function using setTimeout().

Callbacks: Don’t make function async

Newton School of Technology

Callbacks don’t make functions asynchronous!

```
function greet(name, callback) {
    console.log("Hello, " + name);
    callback();
}

function sayGoodbye() {
    console.log("Goodbye!");
}

greet("Alice", sayGoodbye);
```

Output:
Hello, Alice
Goodbye!

Here, sayGoodbye() executes immediately after console.log(). There's nothing asynchronous happening.

Callbacks: Handle async operations

Newton School of Technology

Let's have a look at end to end comparison of performing async operations without and with callbacks.

```javascript
function getUserData() {
    fetch("https://jsonplaceholder.typicode.com/users/1")
      .then(response => response.json());
}

let user = getUserData();
console.log(user);  // ❌ Output: undefined
```

Can you tell why??

Callbacks: Handle async operations

Newton School of Technology

Here fetch() does return a value, but since we are not using callbacks, we can't add a task which only needs to happens after fetch operations completes.

```
function getUserData() {
    fetch("https://jsonplaceholder.typicode.com/users/1")
      .then(response => response.json());
}

let user = getUserData();
console.log(user);  // ❌ Output: undefined
```

We expect to get a value immediately after calling getUserData(), but since fetch() is asynchronous, the data isn't available yet, so we get undefined.

Callbacks: Handle async operations

Newton School of Technology

Instead we need to pass a function to which gets invoked only after fetch() completes its execution. And that function is called **Callback Function**.

```javascript
function getUserData(callback) {
    fetch("https://jsonplaceholder.typicode.com/users/1")
      .then(response => response.json()) // Parse JSON response
      .then(data => callback(data)) // Pass data to callback
      .catch(error => console.error("Error fetching data:", error));
      // Handle errors
}

console.log("Fetching user data...");

getUserData(
    (user) => {
        console.log("User data received:", user);
    }
);

console.log("Waiting for response...");
```

Callback function got received

Callback function gets invoked only when fetch operation completes

Passing the callback function to getUserData() function

Challenge with Callbacks

Newton School of Technology

Callbacks facilitate asynchronous operations, but when multiple operations need to be performed sequentially, they can lead to callback hell.

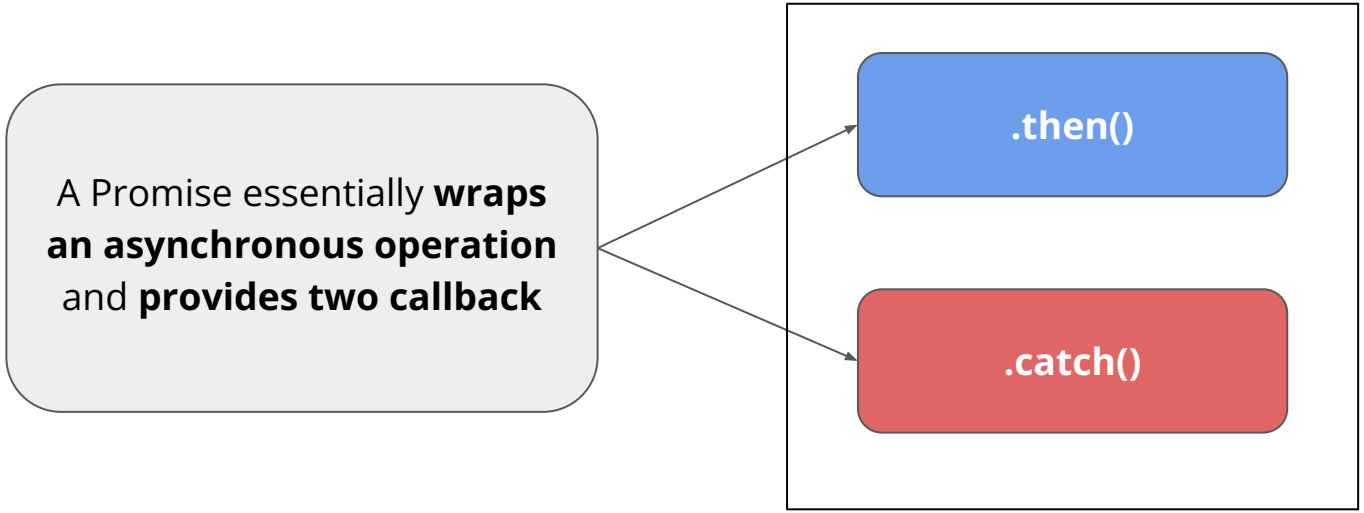
```
a(function (resultsFromA) {
    b(resultsFromA, function (resultsFromB) {
        c(resultsFromB, function (resultsFromC) {
            d(resultsFromC, function (resultsFromD) {
                e(resultsFromD, function (resultsFromE) {
                    f(resultsFromE, function (resultsFromF) {
                        console.log(resultsFromF);
                    })
                })
            })
        })
    })
});
```

Callback hell makes code hard to debug and update, leading to poor scalability.

Promises: a more structured way

Newton School of Technology

Promises are built on callbacks, but they offer a more structured and manageable way to handle asynchronous operations compared to using callbacks directly.



Callbacks **chained with promise**

Difficulty with promises

Newton School of Technology

Promises are a huge improvement over callbacks but Similar to callback hell, if you nest too many .then() blocks, the code can become difficult to manage and read.

```
// Chaining multiple promises in a complex sequence
getUserInfo(1)
  .then(user => {
    return getUserOrders(user.userId);
  })
  .then(orders => {
    return getOrderDetails(orders[0]);  // Details for the first order
  })
  .then(orderDetails => {
    return getItemReviews(orderDetails.items[0]);  // Reviews for the first item
  })
  .then(reviews => {
    return processReview(reviews[0]);  // Process the review for the first item
  })
  .then(processedReview => {
    return getItemReviews(102);  // Reviews for the second item in the order
  })
  .then(reviews2 => {
    return processReview(reviews2[0]);  // Process the second review
  })
  .then(processedReview2 => {
    return getOrderDetails(orders[1]);  // Now for the second order
  })
```

As you keep chaining more .then() calls, the code starts getting increasingly indented. This makes it difficult to read, track, and debug.

How to improve code with async/await

Newton School of Technology

The solution to this is to use async/await, which simplifies the syntax and eliminates the need for deep chaining.

```javascript
async function processUserData() {
    try {
      const user = await getUserInfo(1);
      const orders = await getUserOrders(user.userId);
      const orderDetails = await getOrderDetails(orders[0]);
      const reviews = await getItemReviews(orderDetails.items[0]);
      const processedReview = await processReview(reviews[0]);
      const reviews2 = await getItemReviews(102);
      const processedReview2 = await processReview(reviews2[0]);
      const orderDetails2 = await getOrderDetails(orders[1]);

    } catch (error) {
      console.log('Error:', error);
    }
}

processUserData();
```

Notice how much easier it is to ***read, update, and debug*** our code now?

Newton School of Technology

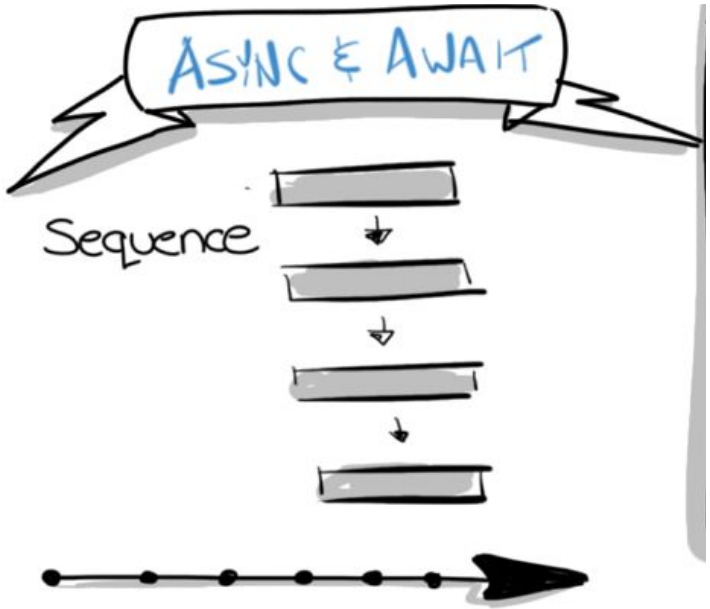
async / await

Solution to deep nesting/chaining

Understanding async / await

Newton School of Technology

async and await are language features in JavaScript that simplify handling asynchronous operations, making the code look and behave more like synchronous code.



While the async function runs asynchronously, the operations inside it execute sequentially.

Basic structure of async / await

Newton School of Technology

Async makes a function return a promise, and await makes the function wait for the promise to finish before moving to the next step.

```
async function() {
  await ...
}
```

We use the await keyword before any task that is asynchronous and takes time to complete, allowing the code to pause and wait for the result.

async / await: Example

Let's understand it with an example:-

Declares a function as asynchronous

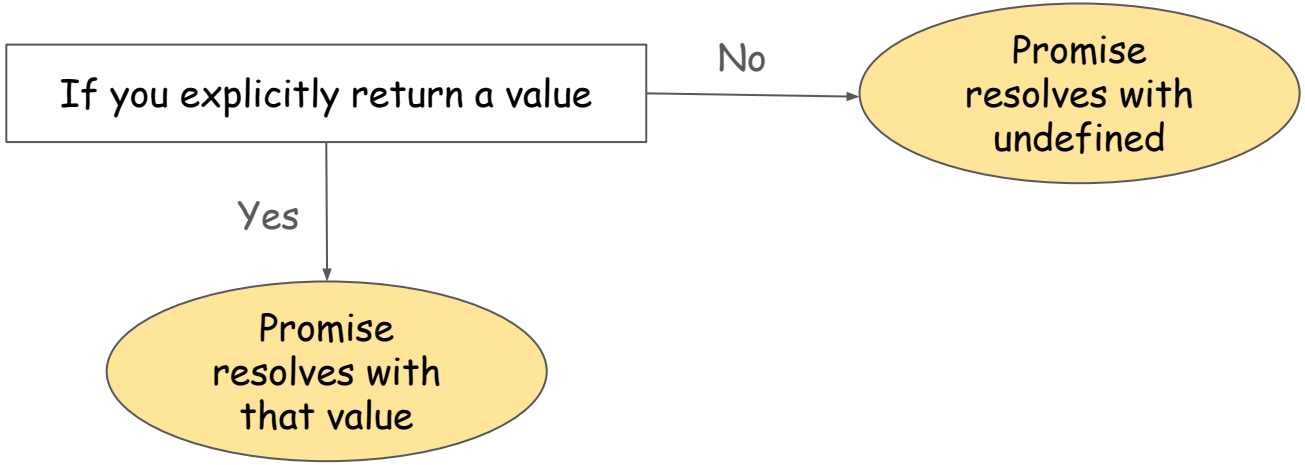
```
// Async function to fetch data from a shorter API URL
async function getDataFromAPI() {
    const response = await fetch("https://jsonplaceholder.typicode.com/todos/1");
    const data = await response.json();  // Parsing the response as JSON
    console.log(data);  // Logging the fetched data
}

getDataFromAPI();
```

Pauses the execution of next statement until response.json() gets resolved

Return values in async/await

Newton School of Technology

An async function always returns a promise. If you return a value, the promise resolves with it; otherwise, it resolves with undefined.



async/await: Explicitly return value

Newton School of Technology

When an async function completes execution and reaches a return statement, the returned value is automatically wrapped in a Promise.

```javascript
// Async function to fetch data from a shorter API URL
async function getDataFromAPI() {
    const response = await fetch("https://jsonplaceholder.typicode.com/todos/1");
    const data = await response.json();  // Parsing the response as JSON
    console.log(data);  // Logging the fetched data
    return data;  // Explicitly returning the fetched data
}

// Calling the function and handling the returned promise
getDataFromAPI().then((data) => {
    console.log("Returned Data:", data);  // Handling the returned value
});
```

Here we are explicitly returning a value

Capturing that value using .then()

async/await: No return value

If the function does not have an explicit return, it implicitly returns undefined wrapped in a Promise

```
// Async function to fetch data from a shorter API URL
async function getDataFromAPI() {
    const response = await fetch("https://jsonplaceholder.typicode.com/todos/1");
    const data = await response.json();  // Parsing the response as JSON
    console.log(data);  // Logging the fetched data
}

const apiReturnedVal = getDataFromAPI();

// Printing value
console.log(apiReturnedVal);
// Output: undefined
```

Here we got undefined because we didn't returned anything implicitly

Newton School of Technology

Error Handling

Using async / await

Error Handling in async / await

When working with async/await, errors can occur due to network failures, invalid responses, or unexpected issues.

To handle errors, we use

Use try/catch in async block

Use catch on the returned promise

try / catch: Saves your day

Imagine a nice day, driving your car and suddenly met with an accident. But don't worry airbags are there to catch and save you.

Car/application running smoothly

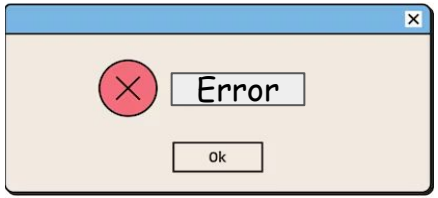
A crash occurred causing an injury/error

Thankfully error got caught not causing much damage

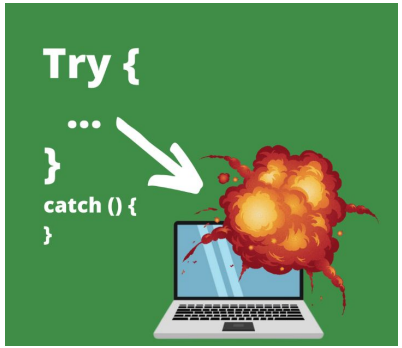
try / catch: Saves your day

Definitely we were not talking about car!!

Web application running smoothly



Error occurred



try...catch caught the error

Error Handling: try / catch

Newton School of Technology

try...catch block in general is used to handle errors. We can use try...catch in the following way:-

```
try {
    let result = 10 / 0;
    console.log("Result:", result);

    let name = undefined;
    console.log(name.length); // This will cause an error
} catch (error) {
    console.log("Oops! Something went wrong:", error.message);
}
```

Code which might throw an error

Capturing the error

Error Handling: try / catch inside async

Newton School of Technology

We can capture errors in similar fashion inside our async function:-

```javascript
1  async function getDataFromAPI() {
2      try {
3          console.log("Fetching data...");
4          const response = await fetch("https://jsonplaceholder.typicode.com/posts/1");
5          if (!response.ok) {
6              throw new Error(`HTTP Error! Status: ${response.status}`);
7          }
8          const data = await response.json();
9          console.log("Data fetched:", data);
10
11         return data; // Explicitly returning the fetched data
12     } catch (error) {
13         // Handling any errors that occur during the fetch process
14         console.log("Error fetching data:", error.message);
15     }
16 }
17
18 getDataFromAPI();
```

Might throw an error

Gets captured in catch block

Error Handling: catch in returned promise

Or else we can attach a catch block in returned promise and capture all the errors there.

```
1  async function getDataFromAPI() {
2      console.log("Fetching data...");
3      const response = await fetch("https://jsonplaceholder.typicode.com/posts/1");
4      if (!response.ok) {
5          throw new Error(`HTTP Error! Status: ${response.status}`);
6      }
7      const data = await response.json();
8      console.log("Data fetched:", data);
9  }
10
11 getDataFromAPI()
12     .catch((error) => {
13         console.log("Error fetching data:", error.message);
14     });
```

Any error occured

Gets captured here

Quiz

Test Your Understanding!

References

1. **MDN Web Docs - JavaScript**: Comprehensive and beginner-friendly documentation for JavaScript.
   https://developer.mozilla.org/en-US/docs/Web/JavaScript
2. **Eloquent JavaScript**: A free online book covering JavaScript fundamentals and advanced topics.
   https://eloquentjavascript.net/
3. **JavaScript.info**: A modern guide with interactive tutorials and examples for JavaScript learners.
   https://javascript.info/
4. **freeCodeCamp JavaScript Tutorials**: Free interactive lessons and coding challenges to learn JavaScript.
   https://www.freecodecamp.org/learn/

Newton School of Technology

Thanks for watching!