Breaking Javascript

Asynchronous

by Shaun Kriel

What is Asynchronous Javascript?

Imagine asking your friend to help you clean your room, but while they are helping, you can go play video games.

You don't have to wait — you can do other stuff at the same time!

That's what asynchronous means:

👉 Do more than one thing without waiting.

Why do we need Asynchronous JavaScript?

- Computers are fast, but some things take time (like getting data from the internet 🎯).
- If JavaScript waited for everything, your screen would freeze! ❄️
- *Asynchronous code lets other things happen while waiting*.

The Three Main Tools

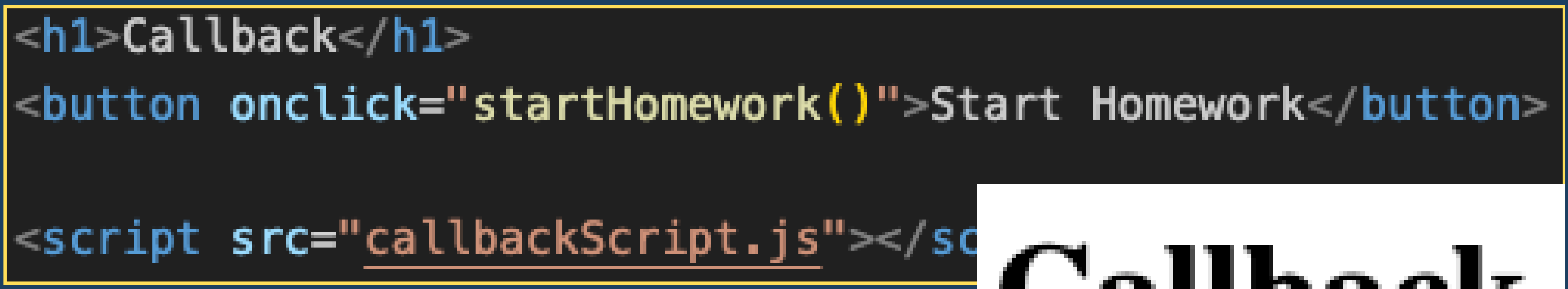
- We can do asynchronous stuff in JavaScript using:
- Callbacks 📞
- Promises 🧹
- Async/Await 💤

Callbacks

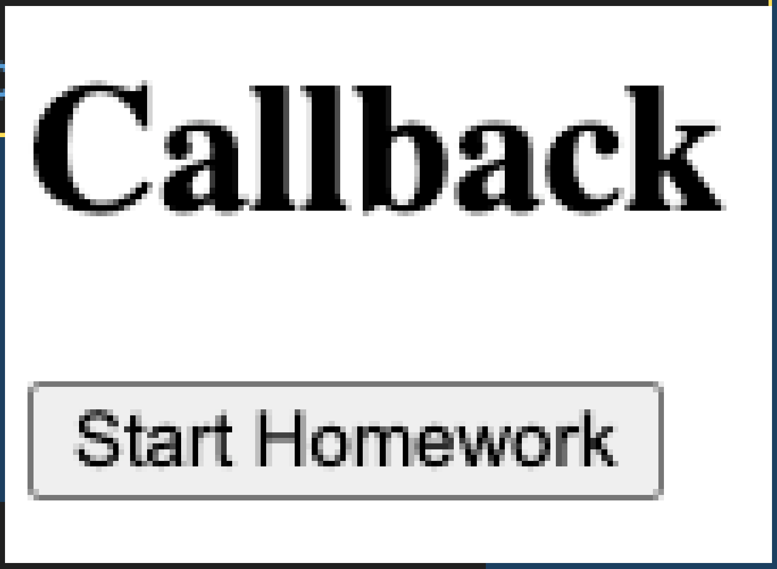
What is a Callback?

Imagine you ask your mom to call you after dinner is ready.
You keep playing. When dinner is ready, she calls you.

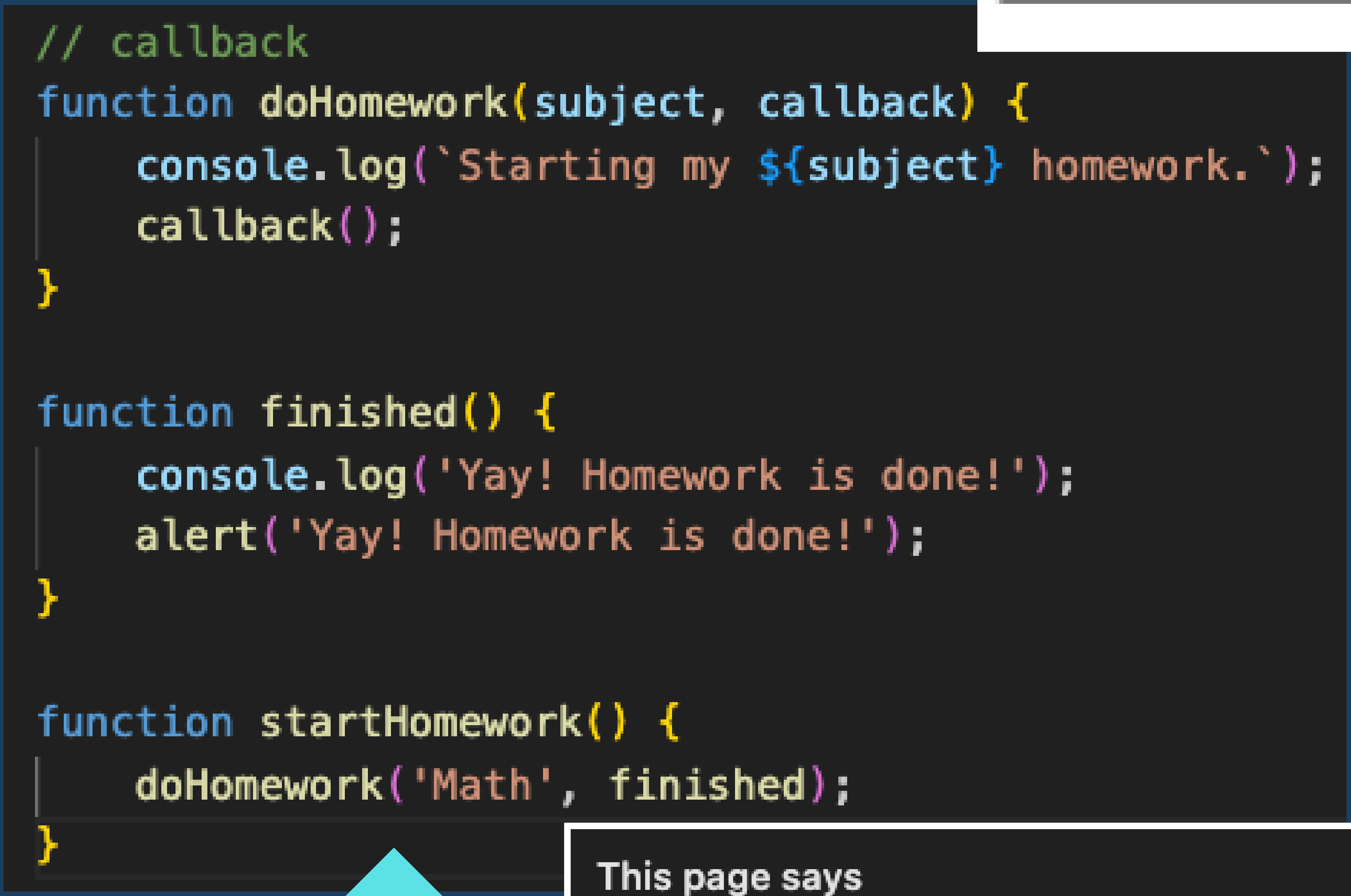
In JavaScript, *a callback is a function that gets called later.*



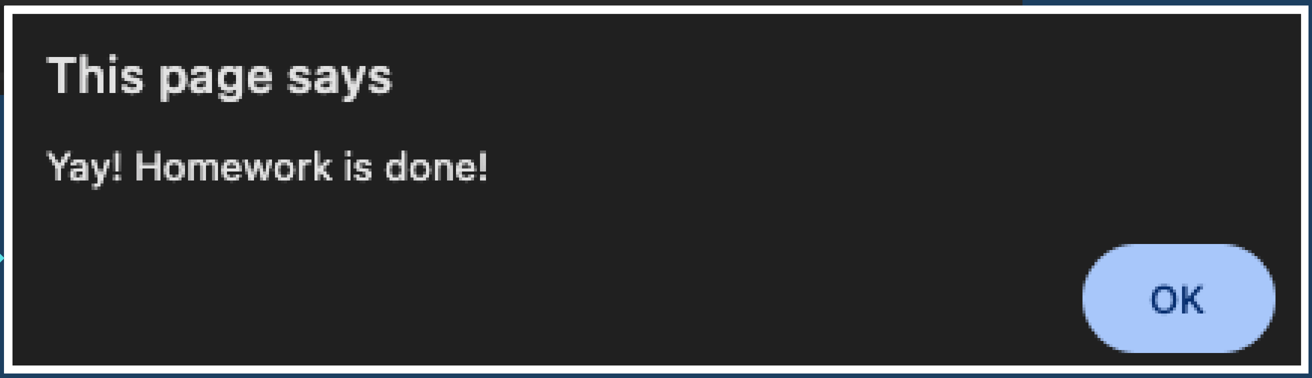
```html
<h1>Callback</h1>
<button onclick="startHomework()">Start Homework</button>

<script src="callbackScript.js"></sc
```



Callback



```javascript
// callback
function doHomework(subject, callback) {
    console.log(`Starting my ${subject} homework.`);
    callback();
}

function finished() {
    console.log('Yay! Homework is done!');
    alert('Yay! Homework is done!');
}

function startHomework() {
    doHomework('Math', finished);
}
```



This page says

Yay! Homework is done!

Callbacks cont.

Callback Example (doHomework)

What Happens:

1. 🖱️ You click the "Start Homework" button.
2. 👉 The **startHomework()** function runs.
3. 📚 **startHomework() calls doHomework('Math', finished)**.
4. 🗣️ Inside **doHomework()**, it:
   - Logs "**Starting my Math homework.**" to the console.
   - **Calls** the **finished()** function that was given as the ***callback.***
5. 🎉 **finished()** runs and:
   - Logs "**Yay! Homework is done!**".
   - Shows an alert popup with "**Yay! Homework is done!**".

Problems with Callbacks 😨

- If you have **lots of callbacks** inside callbacks, it looks **messy**.
- It's called "callback hell".
- Looks like a ladder falling over! 😵‍💫

Promises

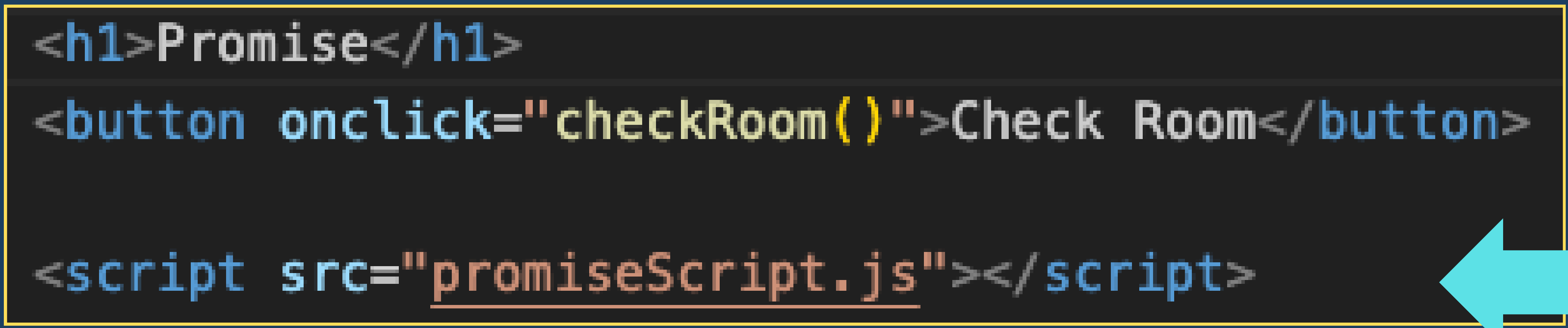
What is a Promise?

🧹 A Promise is like promising your teacher you'll finish your project:

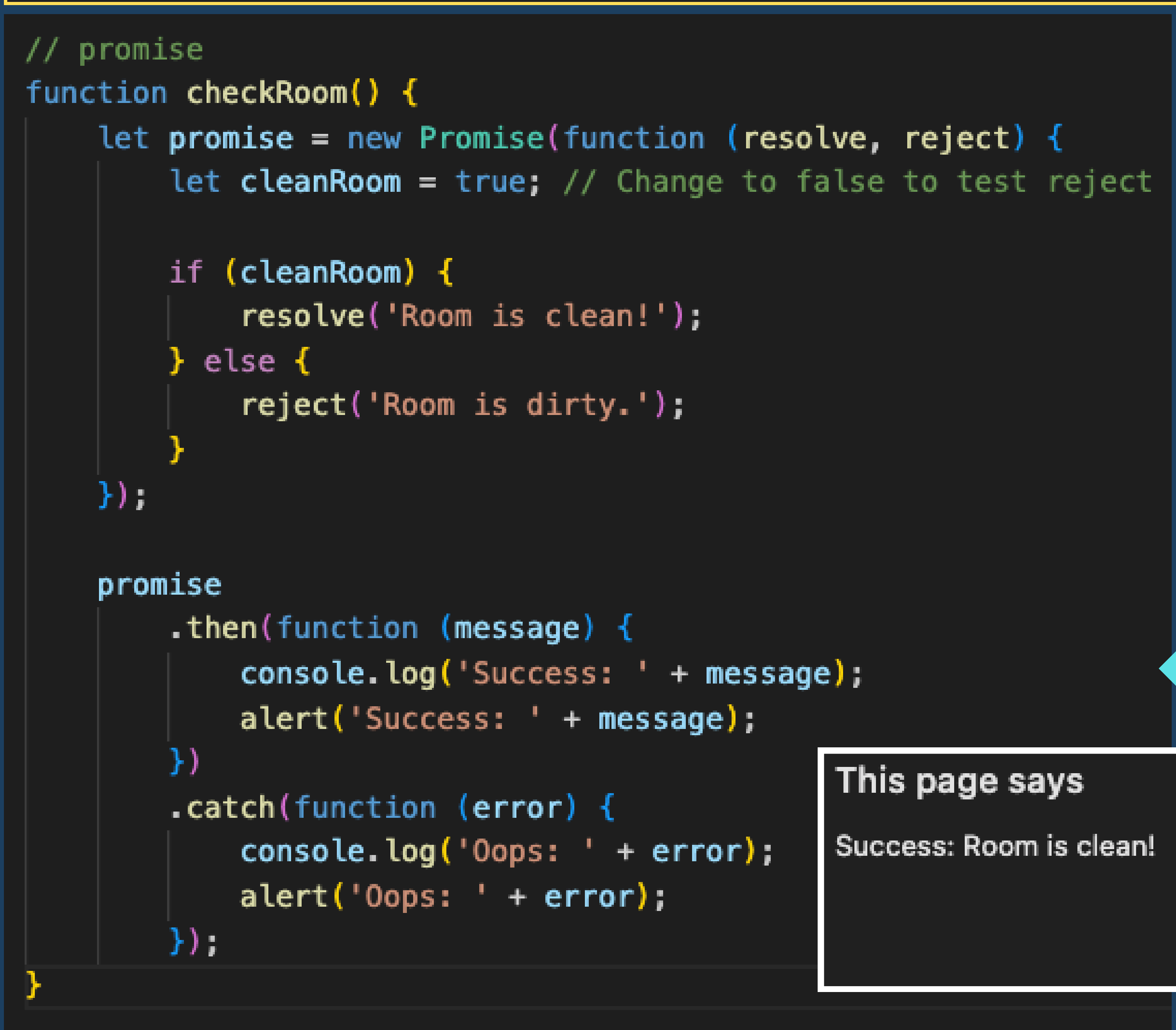
- If you finish, you get a 🎉.
- If you don't, you get 😭.

Promises can be:

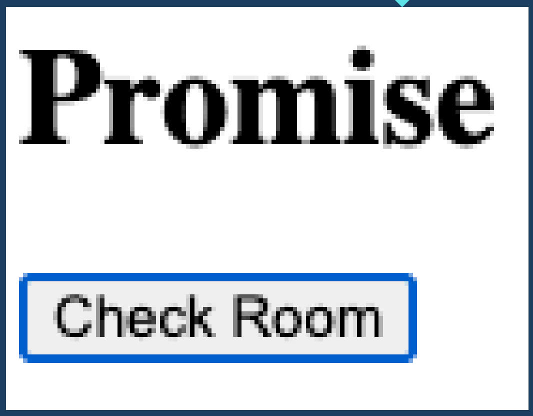
- Pending (not finished)
- Fulfilled (success)
- Rejected (failure)



```html
<h1>Promise</h1>
<button onclick="checkRoom()">Check Room</button>

<script src="promiseScript.js"></script>
```

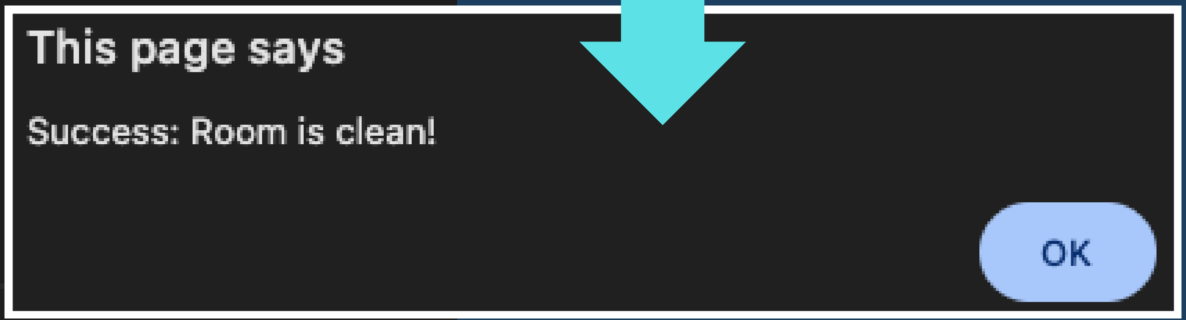


```javascript
// promise
function checkRoom() {
    let promise = new Promise(function (resolve, reject) {
        let cleanRoom = true; // Change to false to test reject

        if (cleanRoom) {
            resolve('Room is clean!');
        } else {
            reject('Room is dirty.');
        }
    });

    promise
        .then(function (message) {
            console.log('Success: ' + message);
            alert('Success: ' + message);
        })
        .catch(function (error) {
            console.log('Oops: ' + error);
            alert('Oops: ' + error);
        });
}
```



Promise

Check Room



This page says

Success: Room is clean!

OK

Promises cont.

Promise Example (cleanRoom Promise)

What Happens:

1. 🖱️ You click the "Check Room" button.
2. 👉 The **checkRoom()** function runs.
3. 🧹 Inside **checkRoom()**, a new Promise is created:
   - **If** cleanRoom is true, it resolves with message "**Room is clean!**".
   - If cleanRoom is false, it rejects with message "**Room is dirty.**".
4. 🌟 Then:
   - If ***resolved,*** it logs and shows "**Success: Room is clean!**".
   - If **rejected**, it logs and shows "**Oops: Room is dirty.**".

Promise chain

- Promises are like **chores**.
- A Promise Chain is doing **one after another.**
- You **wait nicely** after each task before moving on.
- If something goes wrong (like spilling soap!), the chain can **catch the mistake**.

Why is it called a "**chain"**?

Because each task hooks onto the next one —

like train carriages pulling each other 🚂🚃🚃🚃!

You finish one -> then move to the next -> then **the next**.

Imagine this:

You have three chores to do after school:

1. 🛏️ Make your bed
2. 🍽️ Wash the dishes
3. 🐶 Feed the dog

But you can't do them all at once.

You have to finish *one before starting the next*.

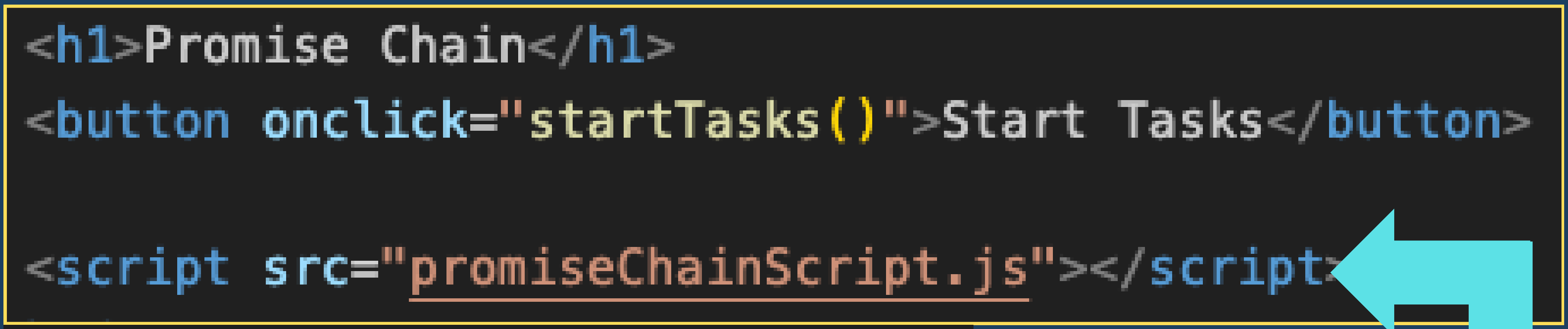
Now, **in code**:

- First, we **Promise** to make the bed.
- When the bed is made, we **Promise** to wash the dishes.
- After the dishes are clean, we **Promise** to feed the dog.

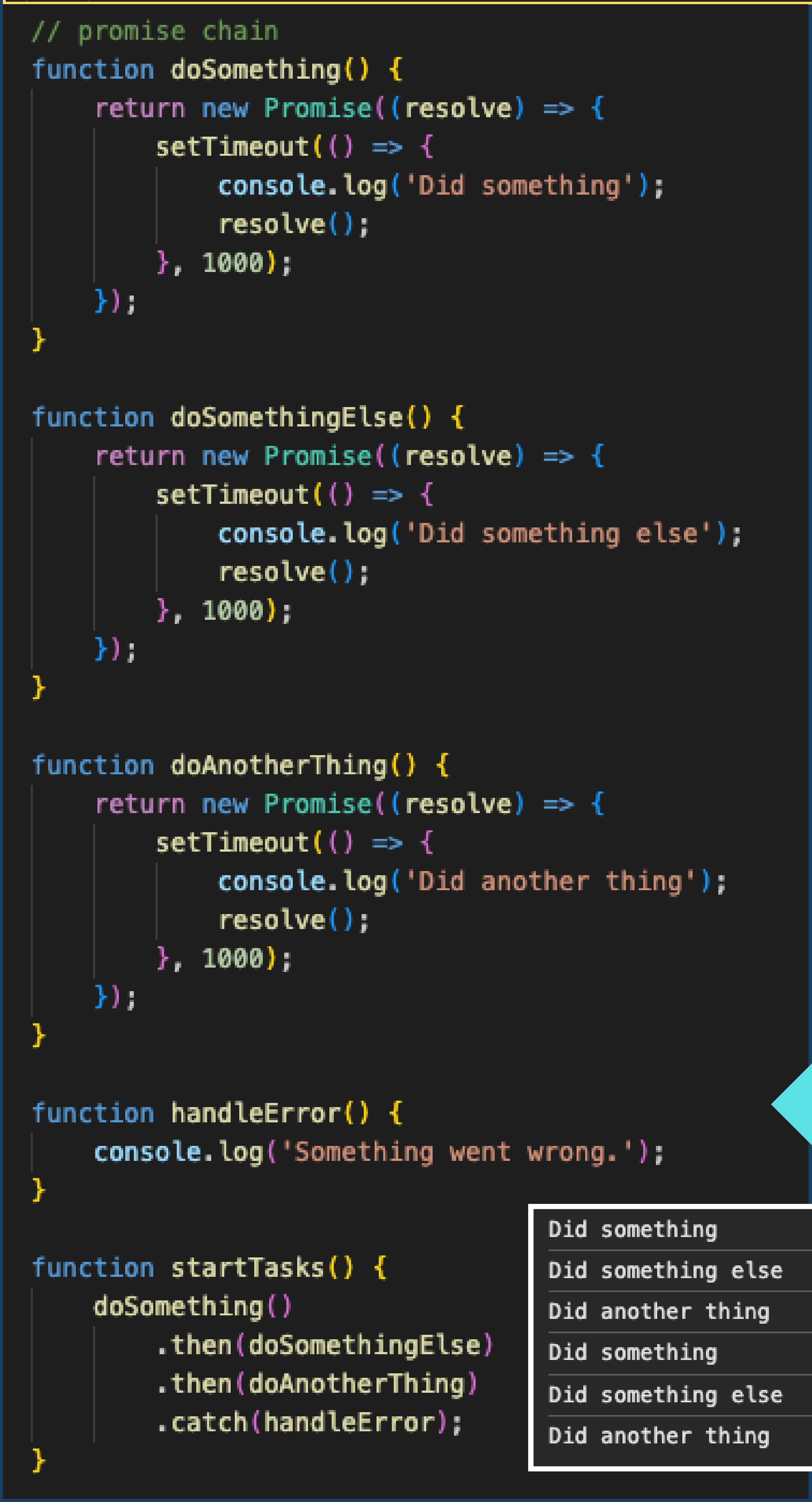
Each Promise says:

"I'll let you know when I'm done, so you can start the next thing!"

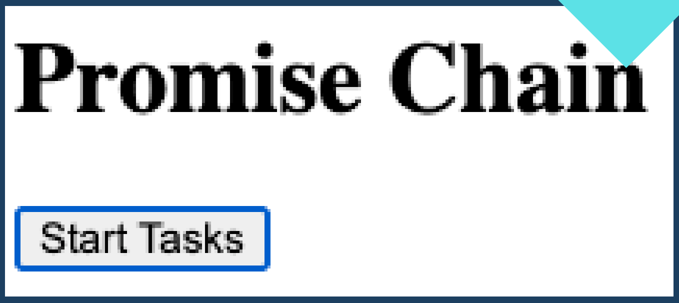
Promise chain cont.

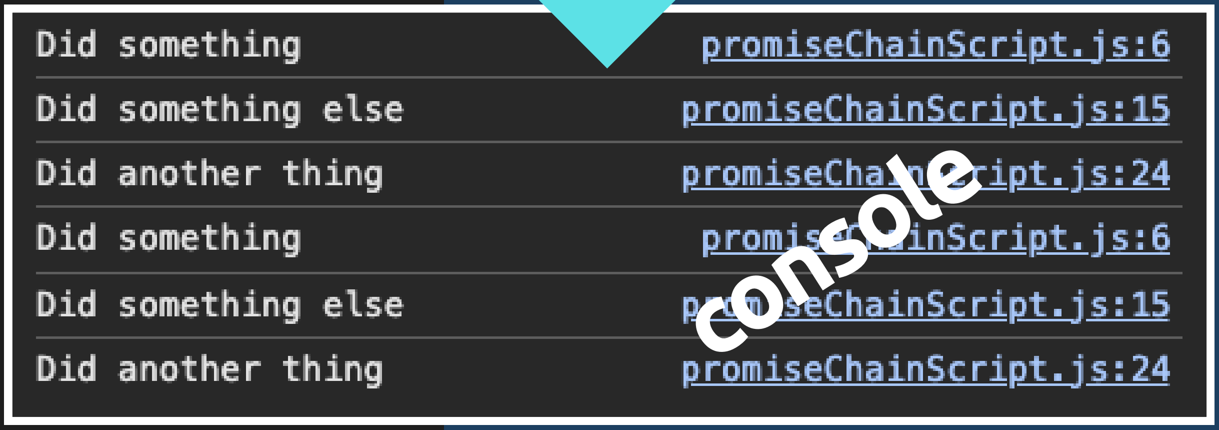


```html
<h1>Promise Chain</h1>
<button onclick="startTasks()">Start Tasks</button>

<script src="promiseChainScript.js"></script>
```



```javascript
// promise chain
function doSomething() {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log('Did something');
            resolve();
        }, 1000);
    });
}

function doSomethingElse() {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log('Did something else');
            resolve();
        }, 1000);
    });
}

function doAnotherThing() {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log('Did another thing');
            resolve();
        }, 1000);
    });
}

function handleError() {
    console.log('Something went wrong.');
}

function startTasks() {
    doSomething()
        .then(doSomethingElse)
        .then(doAnotherThing)
        .catch(handleError);
}
```





```
Did something                promiseChainScript.js:6
Did something else           promiseChainScript.js:15
Did another thing            promiseChainScript.js:24
Did something                promiseChainScript.js:6
Did something else           promiseChainScript.js:15
Did another thing            promiseChainScript.js:24
```

Promise chain cont.

Promise Chain Example (Task Steps)

What Happens:

1. 🖱️ You click the "Start Tasks" button.
2. 👉 The **startTasks()** function runs.
3. 🪜 Step-by-step:
   - **doSomething()** runs: After 1 second, logs "Did something".
   - **doSomethingElse()** runs next: After another 1 second, logs "Did something else".
   - **doAnotherThing()** runs next: After another 1 second, logs "Did another thing".
4. 🚑 If any step fails, it would jump to handleError().

It's like walking up stairs — one step at a time!

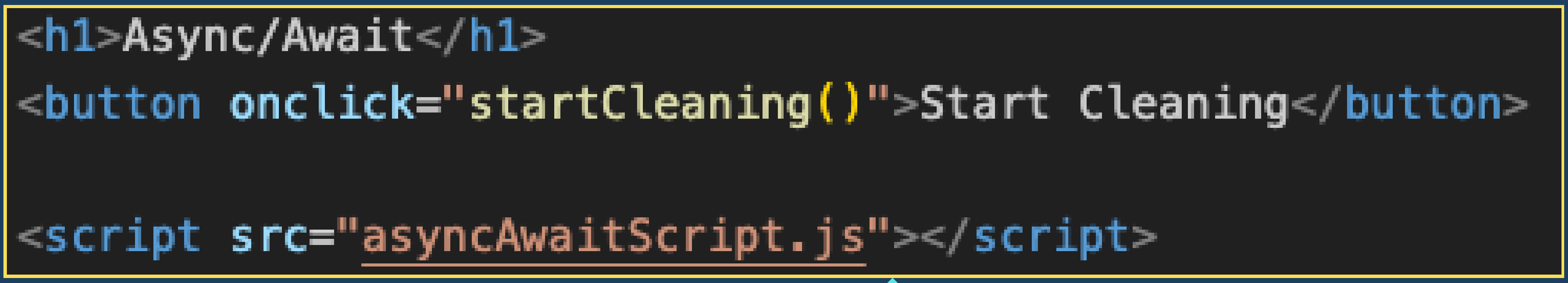
Async/ Await

What is Async/Await?

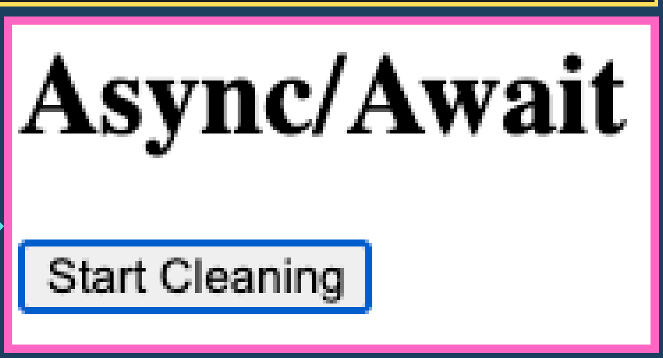
Instead of making promises look messy, we can use async and await to make it look like *normal code but still be asynchronous*!

async = makes a function return a promise.

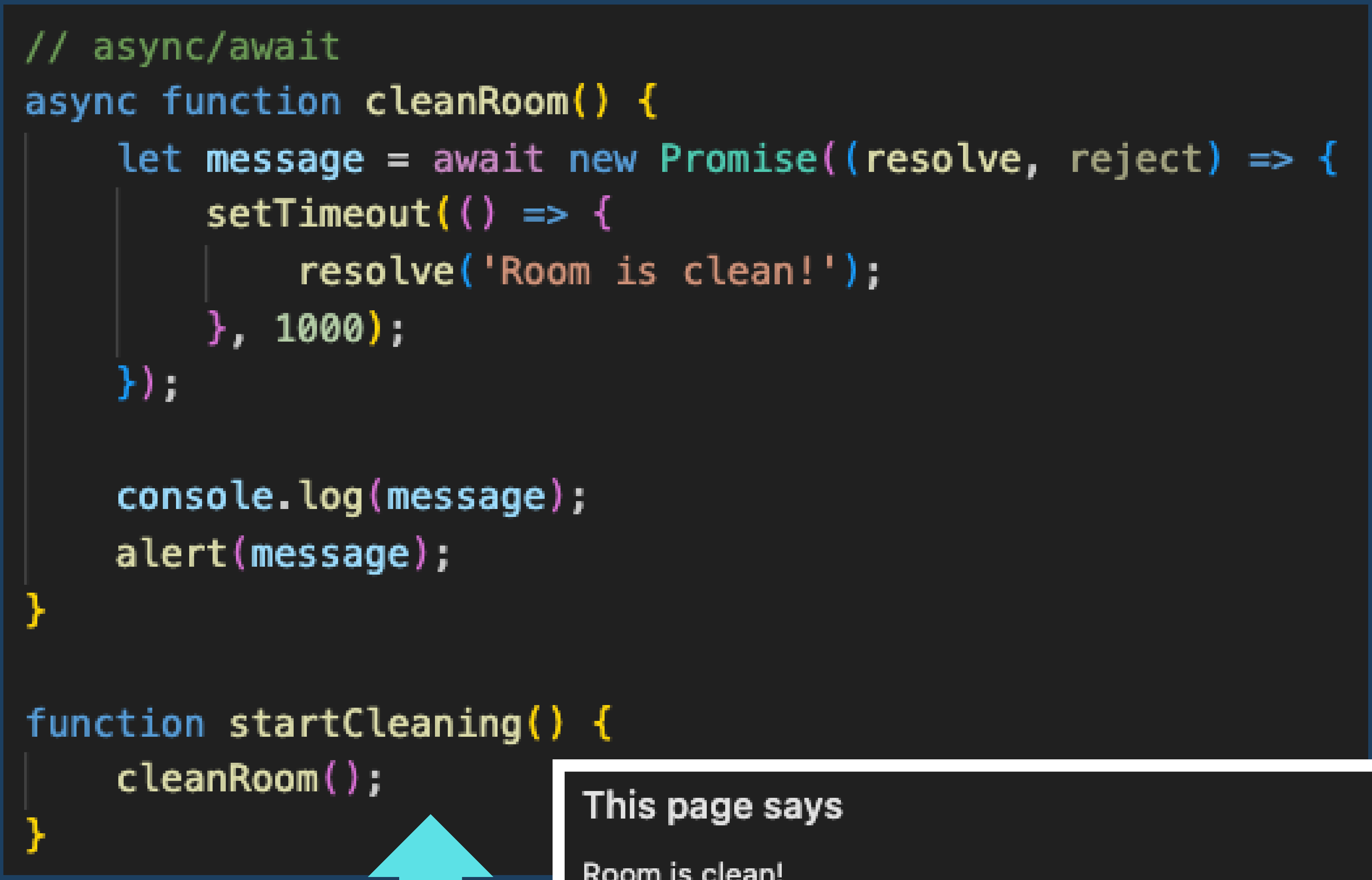
await = wait nicely for the promise to finish.



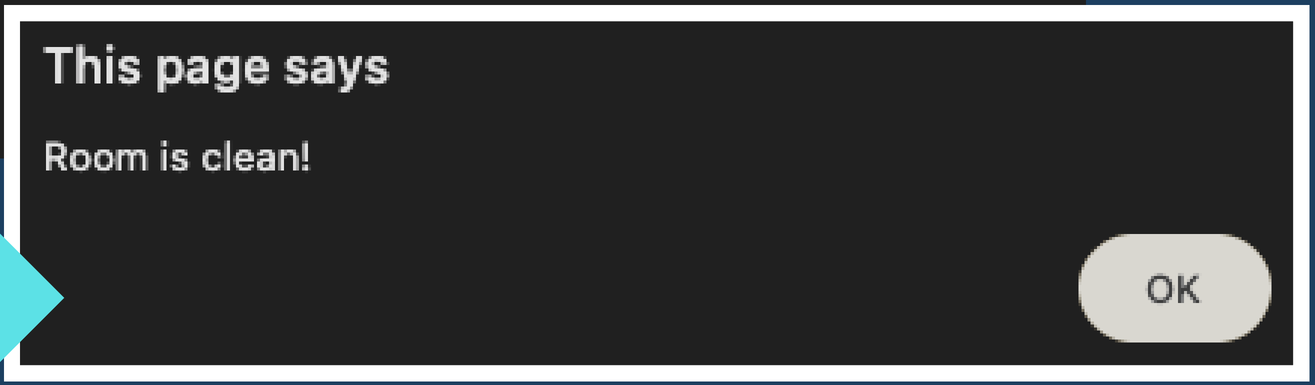
```html
<h1>Async/Await</h1>
<button onclick="startCleaning()">Start Cleaning</button>

<script src="asyncAwaitScript.js"></script>
```



Async/Await



```javascript
// async/await
async function cleanRoom() {
    let message = await new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('Room is clean!');
        }, 1000);
    });

    console.log(message);
    alert(message);
}

function startCleaning() {
    cleanRoom();
}
```



This page says

Room is clean!

Async/ Await cont.

Async/Await Example (cleanRoom async)

What Happens:

1. 🖱️ You click the "Start Cleaning" button.
2. 👉 The **startCleaning()** function runs.
3. 🧹 Inside **startCleaning()**, it calls the **cleanRoom()** function.
4. 🔄 Inside **cleanRoom()**:
    - **await** waits 1 second for a **Promise** to resolve with message "Room is clean!".
5. 🎉 After 1 second:
    - Logs and shows "Room is clean!".

It looks like "*normal*" top-to-bottom code but still waits nicely.

Why Use Async/Await?

✅ **Easier to read**

✅ **Looks like normal code**

✅ **Handles errors nicely**

Async/ Await with Error Handling

What is Async/Await with Error Handling?

Imagine you are waiting for something to happen, like baking cookies. 🍪

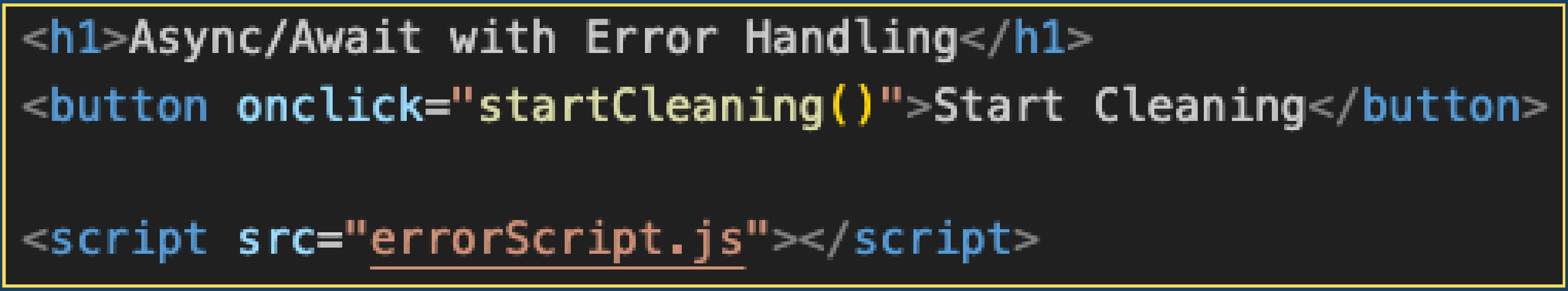
- You put cookies in the oven and wait (await) for them to bake.
- Sometimes, everything goes right — cookies bake perfectly! 🎉
- But sometimes, something goes wrong — like you burn the cookies! 🔥🍪

You don't want to just stand there sad —
you want to **catch** the problem and do something about it (like bake new ones!)

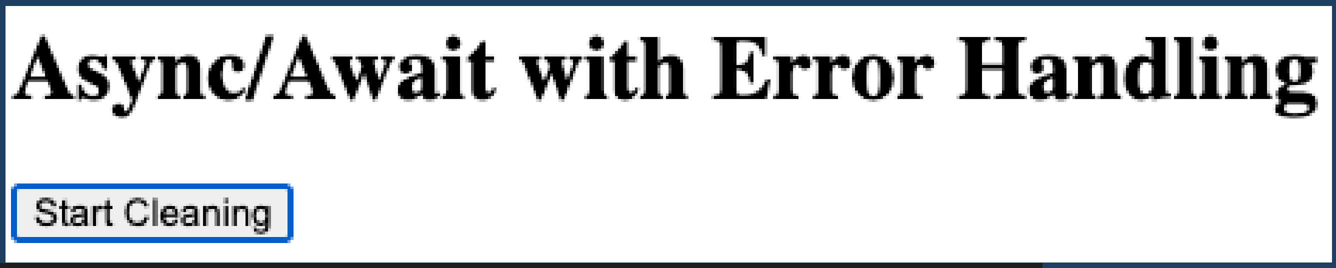
In programming:

- async/await means:
  "**Wait nicely for something slow to finish.**"
- try/catch means:
  "**Try to do it, but if something breaks, catch the error and fix it.**"

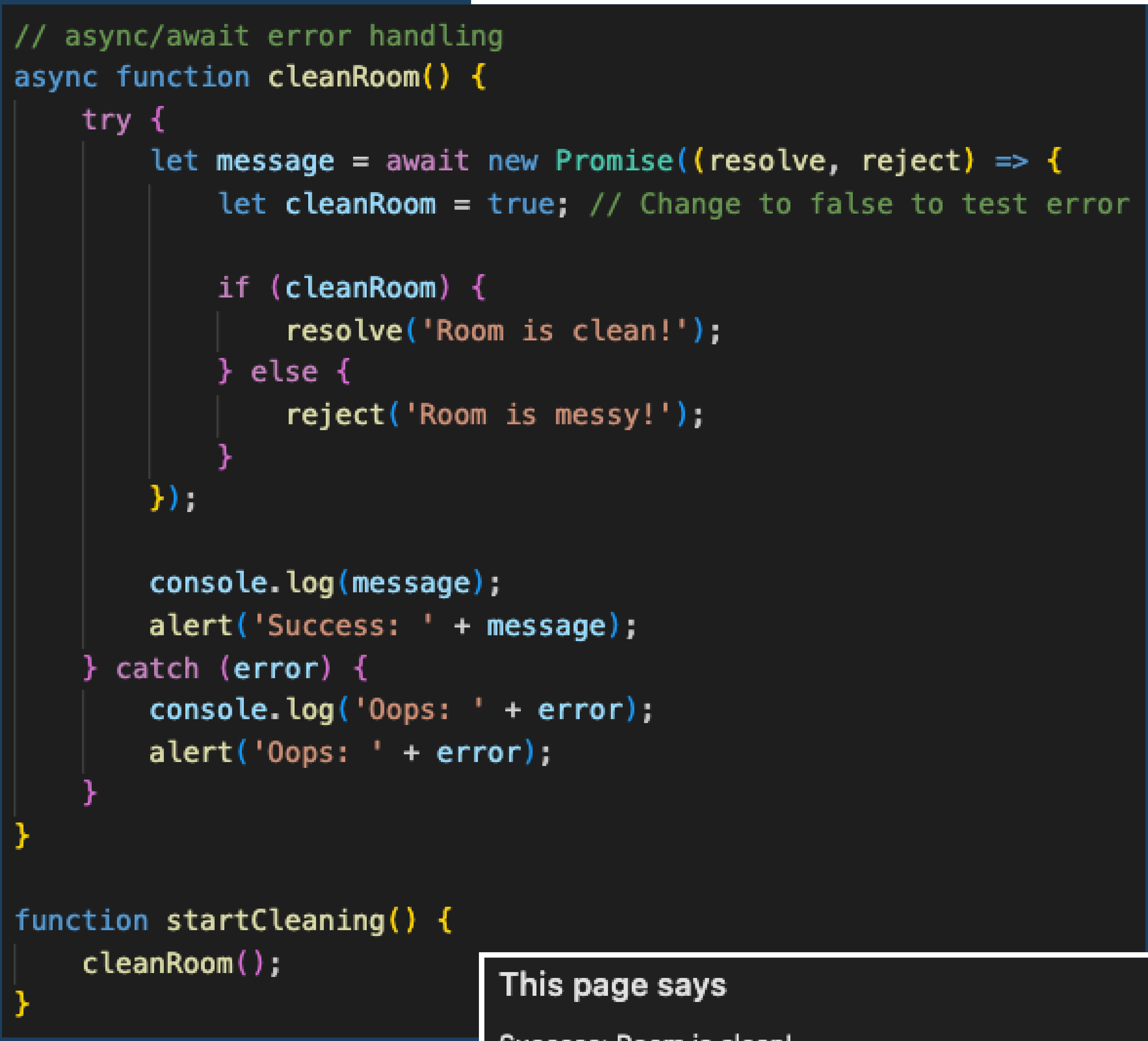
Async/ Await with Error Handling cont.



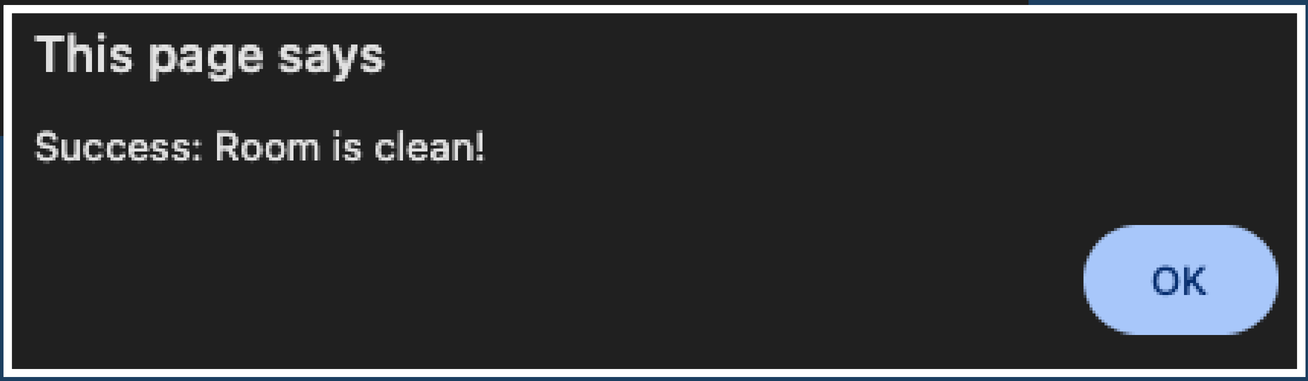
```html
<h1>Async/Await with Error Handling</h1>
<button onclick="startCleaning()">Start Cleaning</button>

<script src="errorScript.js"></script>
```



Async/Await with Error Handling



```javascript
// async/await error handling
async function cleanRoom() {
    try {
        let message = await new Promise((resolve, reject) => {
            let cleanRoom = true; // Change to false to test error

            if (cleanRoom) {
                resolve('Room is clean!');
            } else {
                reject('Room is messy!');
            }
        });

        console.log(message);
        alert('Success: ' + message);
    } catch (error) {
        console.log('Oops: ' + error);
        alert('Oops: ' + error);
    }
}

function startCleaning() {
    cleanRoom();
}
```



This page says

Success: Room is clean!

Async/ Await with Error Handling cont.

Async/Await with Error Handling (try/catch)

What Happens:

1. 🖱️ You click the "Start Cleaning" button.
2. 👉 The **startCleaning()** function runs.
3. 🧹 Inside **startCleaning()**, it calls the **cleanRoom()** function.
4. 🛑 Inside **cleanRoom()**:
    - A Promise checks if cleanRoom = true.
    - If true, await gets "Room is clean!".
    - If false, await throws an error with "Room is messy!".
5. 🧩 The **try** block:
    - Shows success if clean.
6. ⚡ The **catch** block:
    - Catches the error and shows the **"Oops!"** message.

It's like a superhero catching a falling rock before it hits the ground!