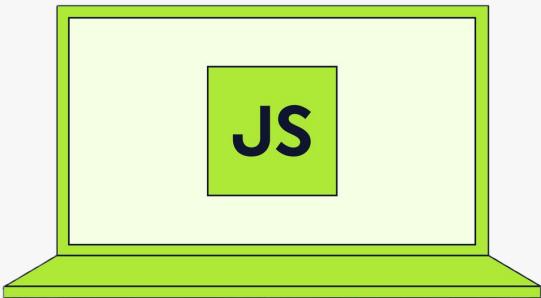




The Complete Javascript Course



@newtonschool

Lecture 10: Introduction to Promises

-Bhavesh Bansal



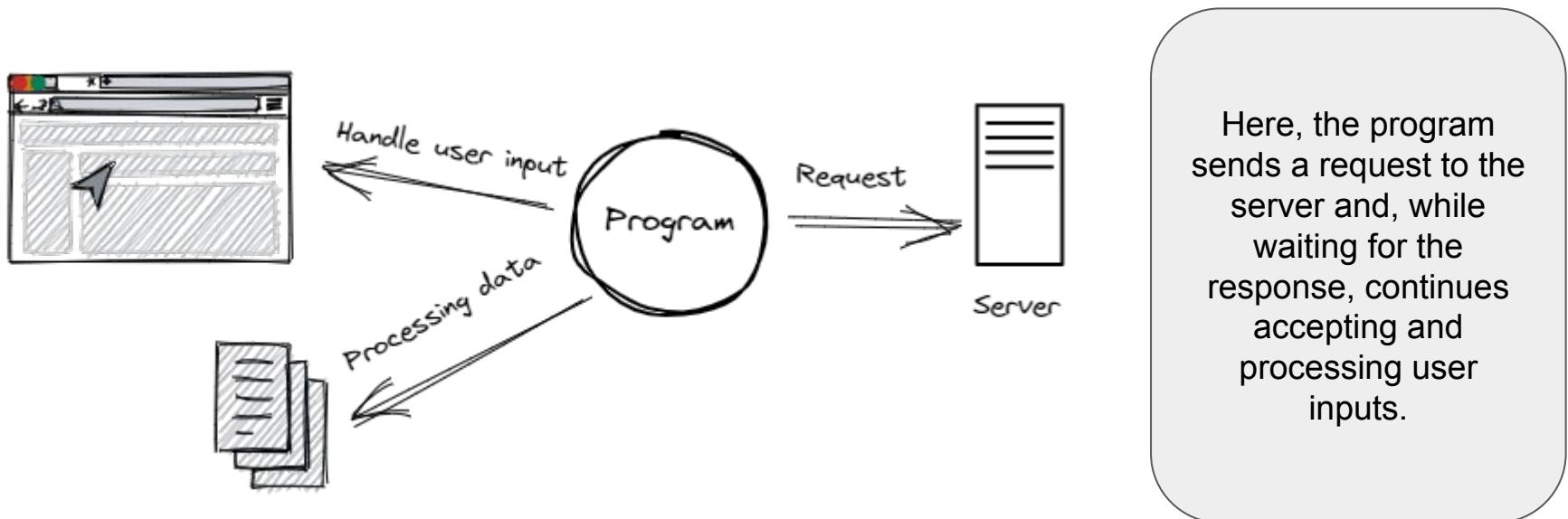
Table of Contents

- Callbacks vs Promise: Need for promise
- Understanding Promises
 - Promises in Javascript
 - Basic Promise Usage
- Creating Promises
 - Promise Constructor
 - Promise Methods: then(), catch() and finally()
- Handling Promise States
 - resolve
 - reject
 - pending
- Error Handling

Why do we need Promises?

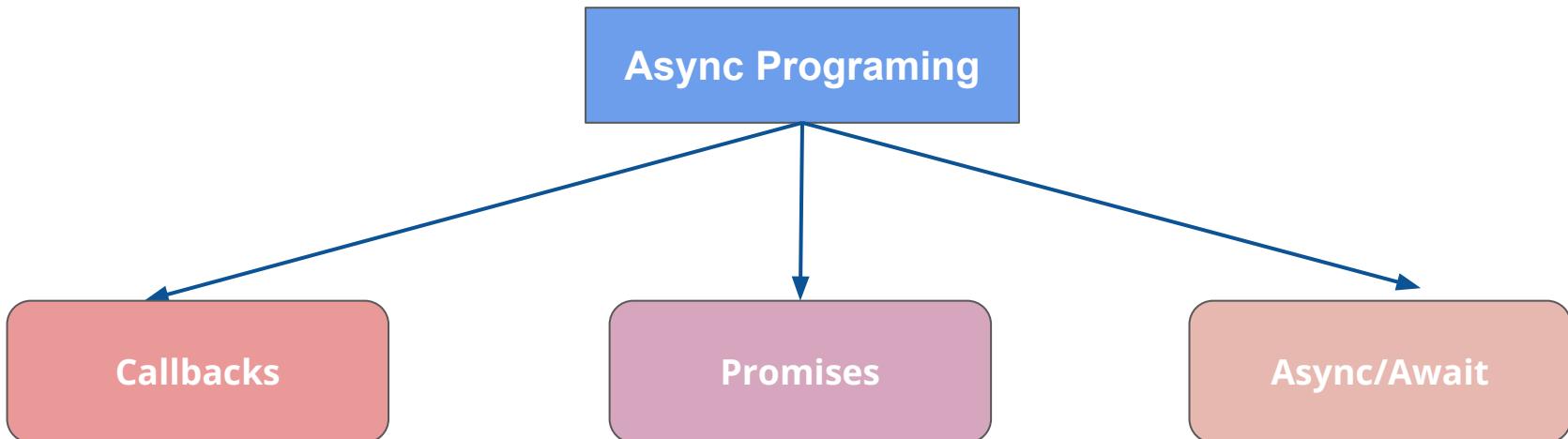
Pre-discussed: Async Programming

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



Pre-discussed: Ways to implement

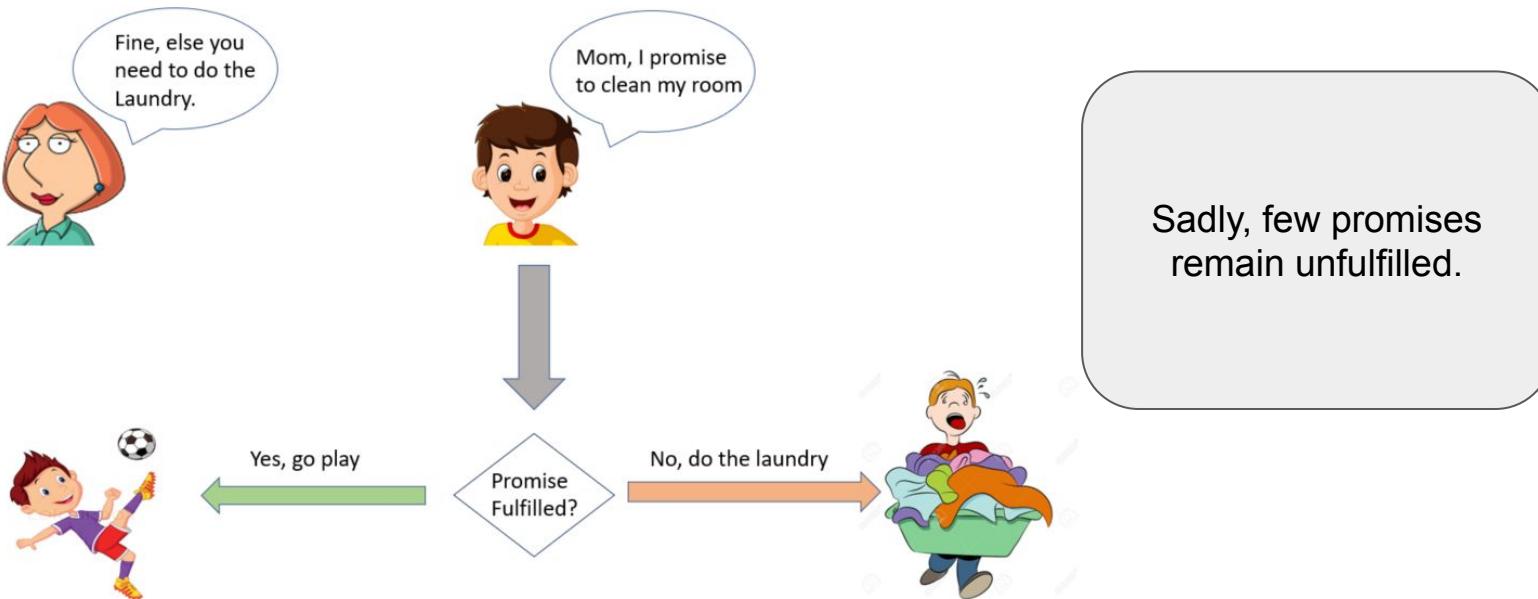
There are three main ways to implement asynchronous programming in JavaScript:



Understanding the Promises

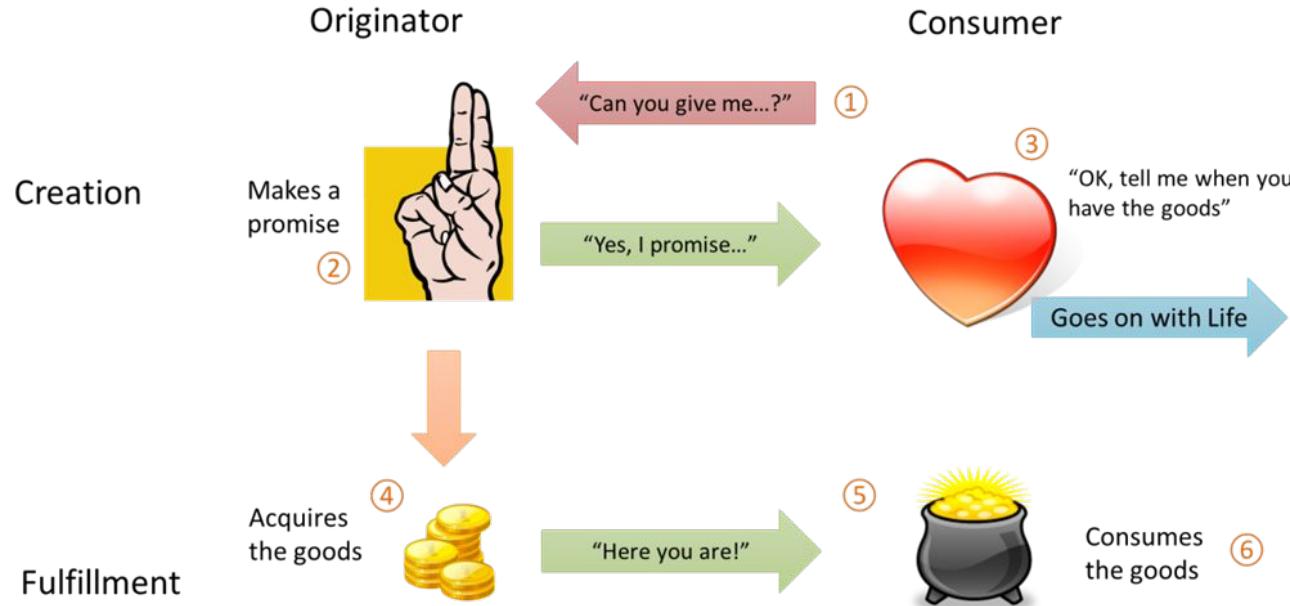
Have you ever made a promise?

If you did, then you must know how valuable promises are. A promise in real life is a commitment or guarantee made by someone.



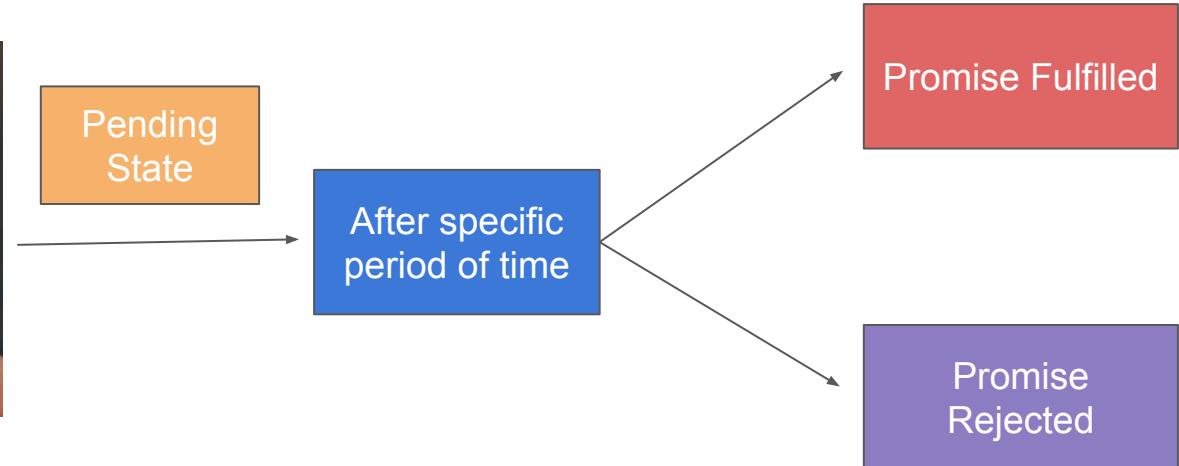
Stages of promise

If you did, then you must know how valuable promises are. A promise in real life is a commitment or guarantee made by someone.



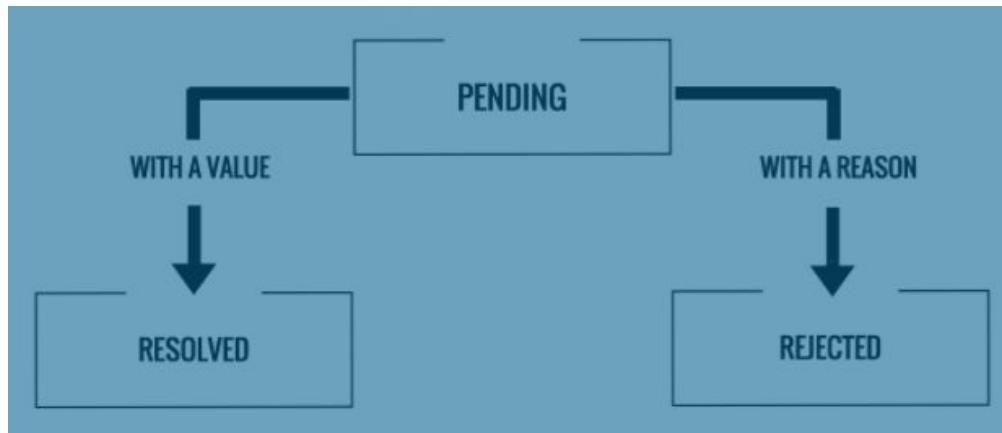
Promises in Javascript

A promise is a commitment in programming—a guarantee that something will happen in the future but not an immediate action. It's a placeholder for future work.



Different promise states

Promises in Javascript exist in three states: pending, fulfilled, and rejected.

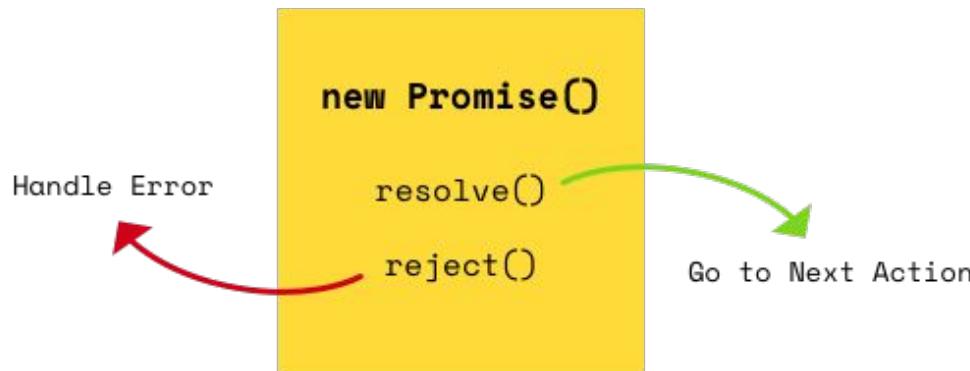


Initially, a promise remains in a pending state, and after some time it either gets resolved/fulfilled or rejected.

Creating Promises

Promise: constructing a promise

To create a promise in JavaScript, you use the `Promise` constructor. The promise constructor receives two functions as arguments: `resolve()` and `reject()`



The `resolve` function is called when the promise is successfully fulfilled.

The `reject` function is called when the promise is rejected.

Promise: Using Promise Constructor

To create a promise in JavaScript, you use the `Promise` constructor. The promise constructor receives two functions as arguments: `resolve()` and `reject()`



```
1 // Constructor to build a promise
2 const myPromise = new Promise([resolve, reject) => {
3     // Body of Promise
4
5     // Here we can either resolve
6     // Or reject promise conditionally
7
8});
```

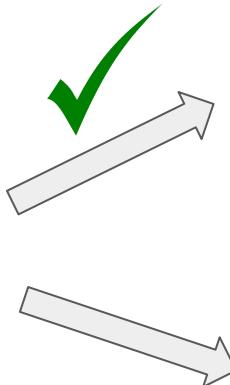
Received resolve and
reject functions as
arguments

Promise: resolve()

The `resolve` method in JavaScript is a static method of the `Promise` object. It is used to create a promise that is resolved with a given value.



Promised to complete the task in meeting



Resolved the task and provided the work data as return value

`resolve()` function's work is to finish the task and provide a return value.



Fail/rejected and return fail/error message

`reject()` function's work is to return an appropriate error message

Promise: resolve()

Let's have a look at a dummy example:



```
1 // Constructor to build a promise
2 const myPromise = new Promise((resolve, reject) => {
3     // Simulation of success in resolving promise
4     const success = true;
5
6     if(success === true){
7         resolve("Promise got resolved");
8     }
9 });

```

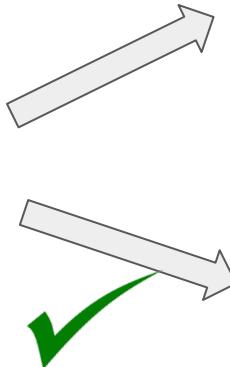
On successful resolution of promise, resolve() returns a value. For simplicity, we are returning a success message.

Promise: reject()

It might be possible that a promise can't be delivered and you need to provide an explanation and, in JavaScript, an error message.



Promised to complete the task
in meeting



Resolved the task and provide
the work data as return value

resolve() function
work is to finish the
task and provide a
return value.



Fail/rejected
and return
fail/error
message

reject() function work
is to return an
appropriate error
message

Promise: reject()

Let's have a look at a dummy example:



```
1 const myPromise = new Promise((resolve, reject) => {
2     // Simulation of success in resolving promise
3     const success = false;
4
5     if(success === true){
6         resolve("Promise got resolved");
7     } else {
8         reject("Promise got rejected");
9     }
10});
```

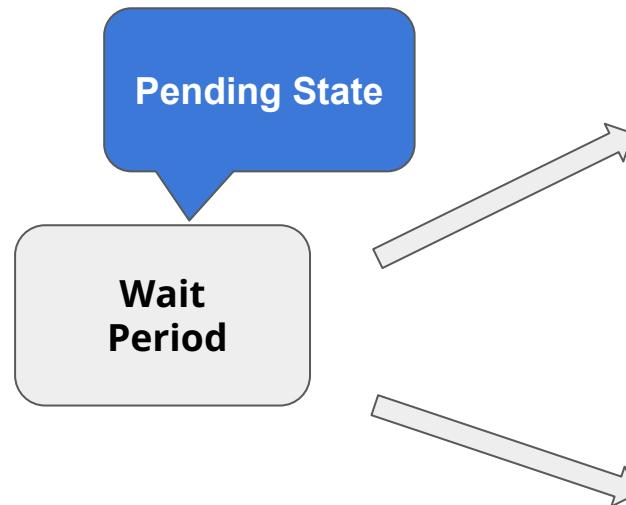
On failure to resolve a promise reject() returns an error message. For simplicity we are returning a failure message..

Promise: pending()

Before either promise gets resolved or rejected, there is some wait period. This wait period is called pending state of promise



Promised to complete the task
in meeting



Resolved the task and provided
the work data as return value



Fail/rejected
and return
fail/error
message

Promise Creation: Example 1

Let's understand promises with a simple example:

```
1 // Constructor to build a promise
2 const myPromise = new Promise((resolve, reject) => {
3     // Asynchronous Operation
4     const success = true; // Simulating a condition
5
6     if(success === true){
7         resolve("Promise got resolved!");
8     } else{
9         reject("Promise got rejected!");
10    }
11});
```



Using `resolve()` to
fulfill the promise



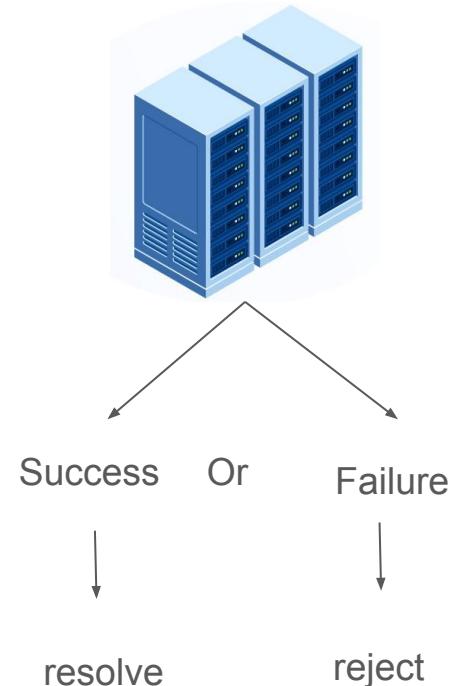
Using `reject()` to
reject the promise

Promise Creation: Example 2

Let's take a more practical example:



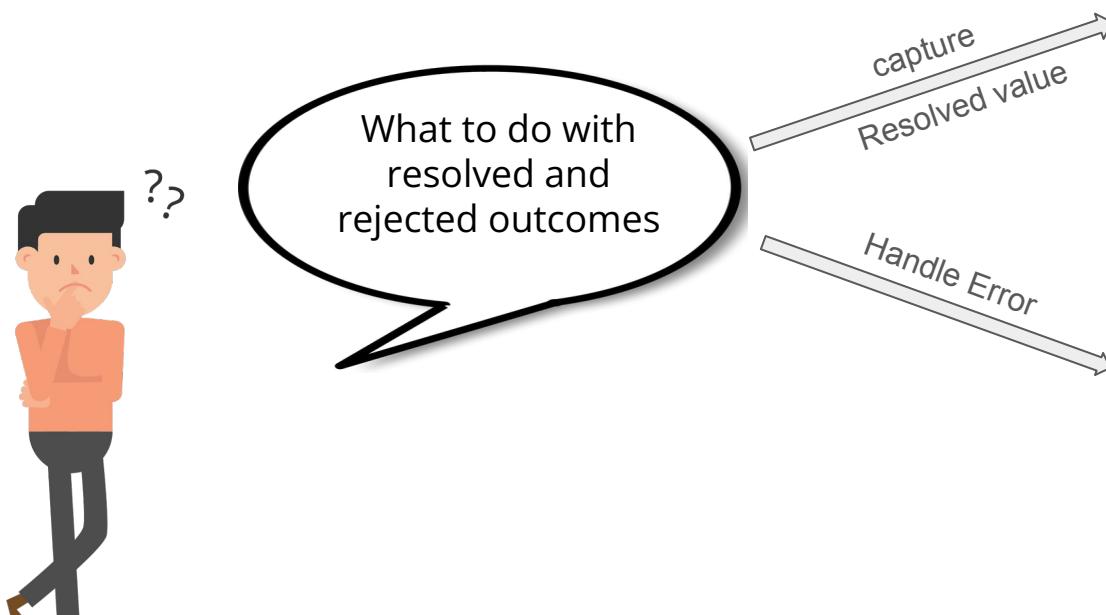
```
1 // Simulating a function that fetches data from a server
2 function fetchDataFromServer() {
3     return new Promise((resolve, reject) => {
4         const serverStatus = false;
5         // Simulate server failure (false means failed)
6
7         if (serverStatus) {
8             resolve('Data fetched successfully');
9         } else {
10             reject('Error: Server is down');
11         }
12     });
13 }
```



Handling the Promise

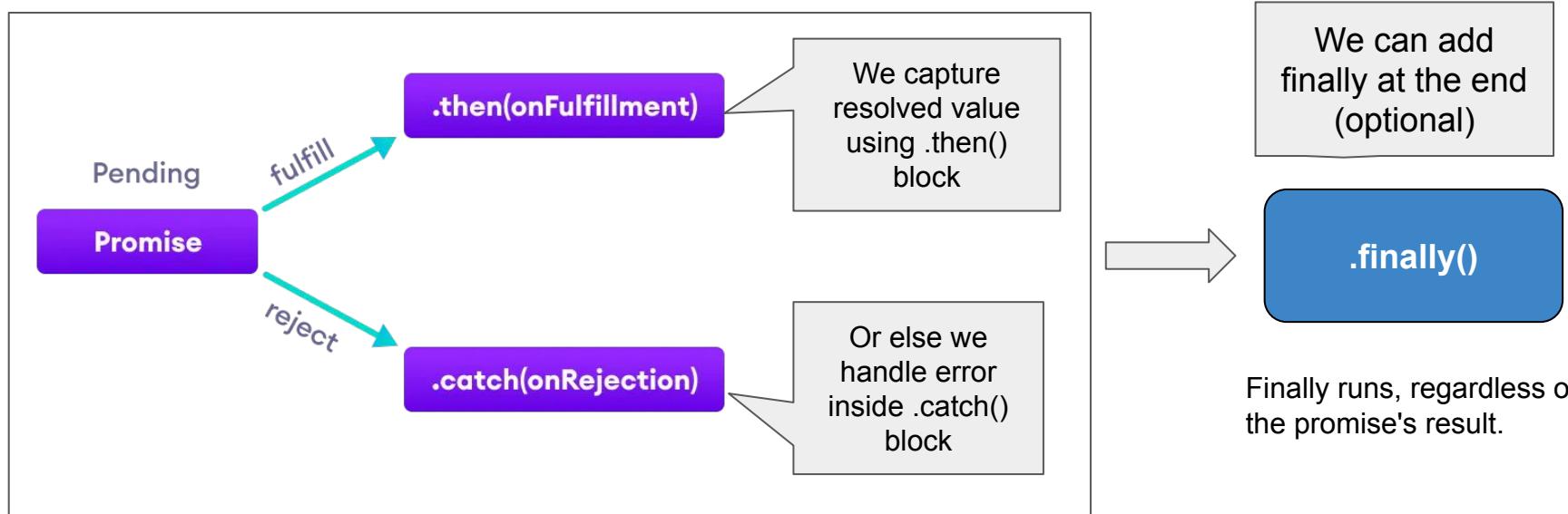
Promises: Handling Promise Outcomes

As we have learned, a promise is initially in a pending state, and then after a while it either gets resolved or rejected.



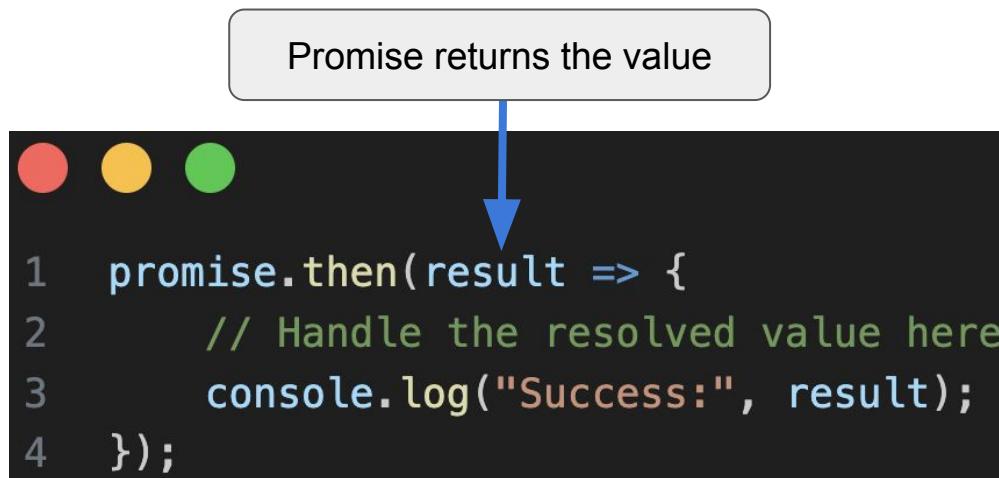
Promises: Bridging Hope and Reality

We hope a promise resolves/fulfilled and gives us a value to work with in the “`then`” block. However, if it gets rejected, we handle it in the “`catch`” block.



Promise: Using the .then() block

The `.then()` block handles data after resolution in a promise chain by capturing it and allowing further processing.



```
promise.then(result => {
  // Handle the resolved value here
  console.log("Success:", result);
});
```

Promise returns the value

We can use the returned value from Promise inside `.then()` block

.then(): Example

The `.then()` block is used to handle the successful resolution of a promise, capturing the returned value and allowing further processing or actions.

```
1 // Simulating a function that fetches data
2 function fetchDataFromServer() {
3     return new Promise((resolve, reject) => {
4         const serverStatus = true;
5         // Simulate server failure (false means failed)
6
7         if (serverStatus) {
8             resolve('Data fetched successfully');
9         } else {
10            reject('Error: Server is down');
11        }
12    });
13 }
```



```
1 // Using .then()
2 fetchDataFromServer()
3     .then(result => {
4         console.log("Success:", result);
5         // Handling the resolved value
6     })

```

Here we are using `.then()` block to capture the resolved value

Promise: Using .catch() block

The `.catch()` block handles errors or rejections in a promise chain, ensuring that any issues during the operation are caught and managed gracefully.



```
1 promise.then(()=>{
2     // then block code
3 }
4     .catch((error)=>{
5         // catch block code
6         console.log("Error: ", error);
7 });


```

`.catch()` is indeed added after the last `.then()` block in most cases to handle any errors that occur while consuming the promise.

.catch(): Example

It might be possible that error occurred while consuming the promise which can be caught using the .catch() block.

```
1 // Using .then()
2 fetchDataFromServer()
3   .then(result => {
4     console.log("Success:", result);
5     // Handling the resolved value
6   })
7   .catch((error) => {
8     console.log("Error: ", error);
9  });
```

Here we are capturing
error while consuming
the promise.

Promise: Using .finally() block

It is possible to chain a block that runs regardless of whether the promise is resolved or rejected using .finally() block.

```
1 promise.then(()=>{
2     // then block code
3 }
4     .catch((error)=>{
5         // catch block code
6         console.log("Error: ", error);
7     })
8     .finally(()=>{
9         // finally block code
10});
```

.finally(): Example

It is possible to chain a block that runs regardless of whether the promise is resolved or rejected using .finally() block.

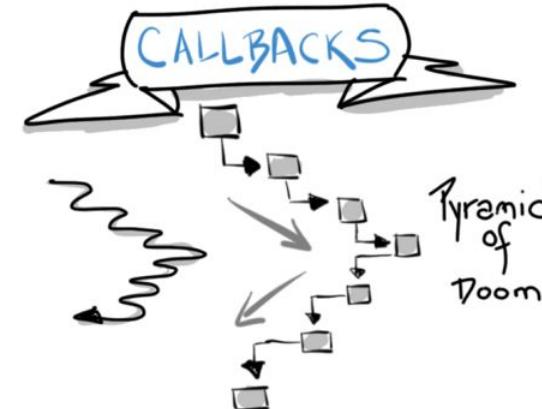
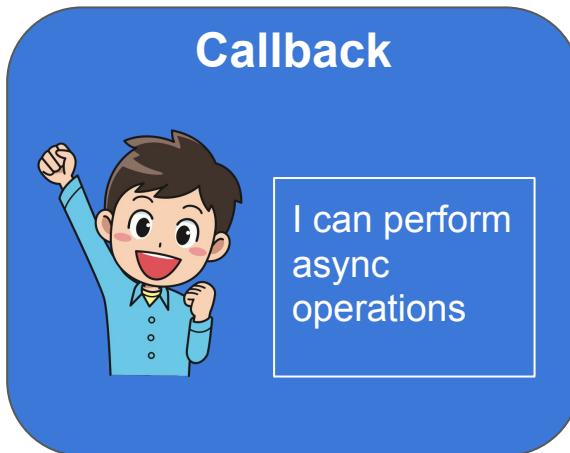
```
1 // Using .then()
2 fetchDataFromServer()
3   .then(result => {
4     console.log("Success:", result);
5     // Handling the resolved value
6   })
7   .catch((error) => {
8     console.log("Error: ", error);
9   })
10  .finally(() => {
11    console.log("Fetch operation completed.");
12    // Executes regardless of success or failure
13 });
```



finally() block

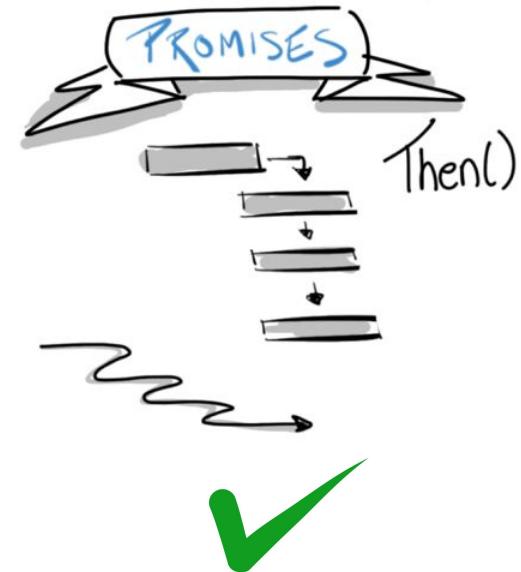
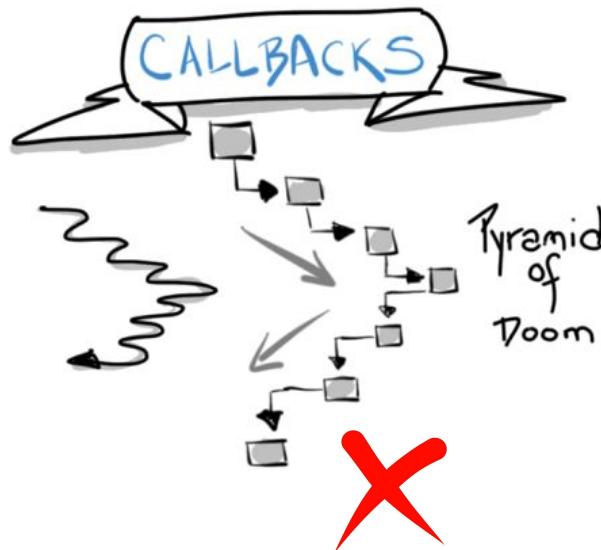
Why do we need Promises??

In the previous lecture we learned that callbacks can be used to implement async programming; then why do we need to make promises?



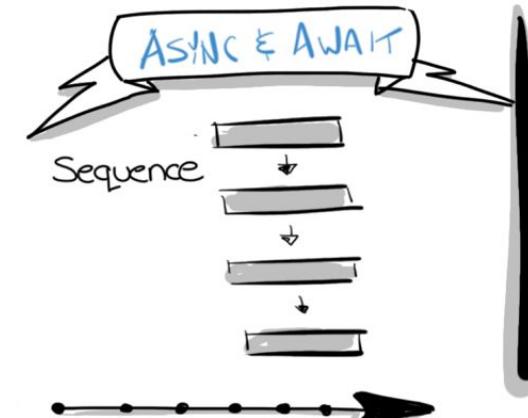
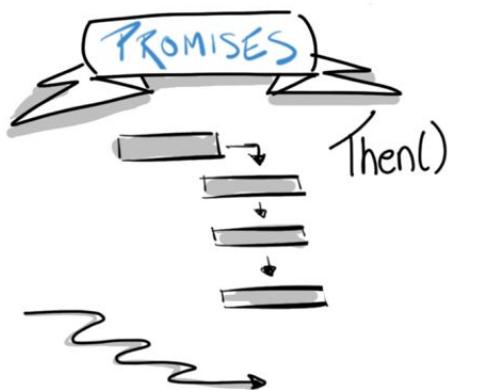
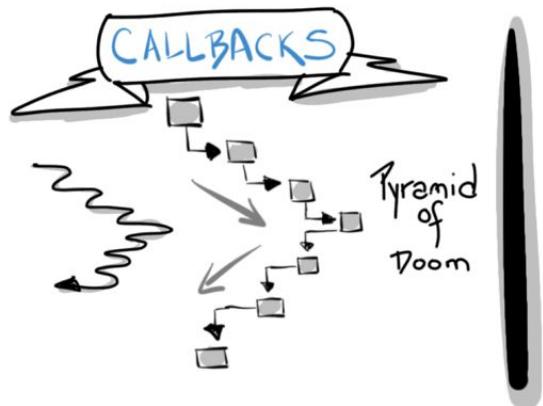
Why do we need Promises??

Callbacks can often lead to cluttered and unreliable code due to nested functions. It is better to chain tasks, where one task starts after the previous one ends, and this is achieved using promises.



Callbacks vs Promises vs Async/await

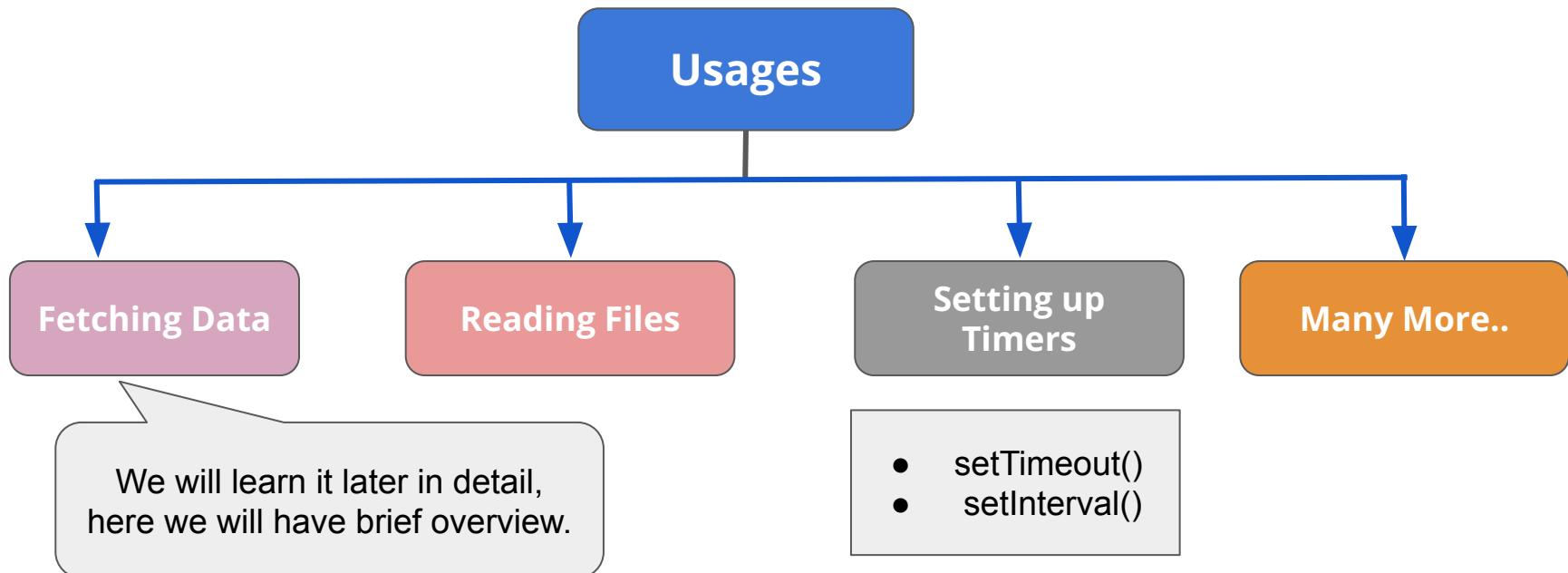
We use callbacks for simple asynchronous tasks, promises for chaining more complex tasks, and `async/await` for writing asynchronous code that appears sequential.



Promise Basic Usages

Basic Promise Usage

Promises in JavaScript simplify handling asynchronous operations. Here are common use cases:



Usage: Fetching Data

We use fetch api to get data from the server.

```
1 // Function to fetch data
2 function fetchData(url) {
3     return new Promise((resolve, reject) => {
4         fetch(url)
5             .then((response) => {
6                 if (!response.ok) {
7                     // On failure, reject the promise
8                     [reject(`HTTP error!: ${response.status}`)];
9                 }
10                return response.json();
11                // Parse JSON from the response
12            })
13            .then((data) => {
14                [resolve(data); // On success return data using resolve]
15            })
16            .catch((error) => {
17                [reject(`Network error: ${error}`)]; // Reject the promise
18            });
19    });
20 }
```

Fetching data from server is an async operation so we used promise for it.

Here we conditionally rejected/resolved the promise

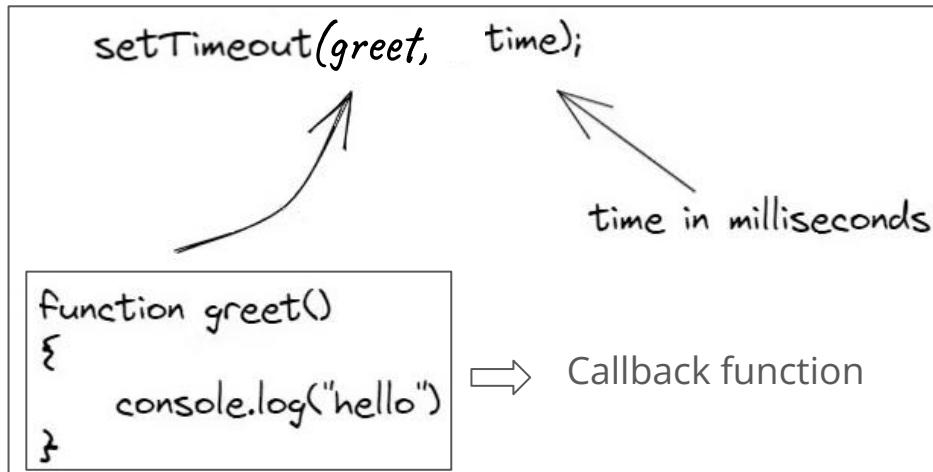
setTimeout and setInterval

with Promises

Pre-discussed: setTimeout

`setTimeout` is a JavaScript function that executes a callback function after a specified delay (in milliseconds).

Syntax:



Pre-discussed: setTimeout Example

We can print specific message after a certain period of time using setTimeout.



```
1 function greet() {  
2     console.log("Hello");  
3 }  
4  
5 // greet function runs after 2 seconds  
6 // i.e. (2000 milliseconds)  
7 setTimeout(greet, 2000);
```

Here we get output after 2000 milliseconds i.e. 2 secs

Output:

Hello

setTimeout wrapped in a Promise

There might be instances where you need to set up timers in JavaScript. In such cases, you can use `setTimeout` in combination with promises

```
1  function delay(ms) {  
2      return new Promise((resolve) => {  
3          setTimeout(resolve, ms);  
4          // Resolve the promise after a delay (in ms)  
5      });  
6  }  
7  
8  delay(2000).then(() => {  
9      console.log("Executed after 2 seconds");  
10 });
```



Creating and returning a promise

Calling the function returning the promise and consuming the returned promise

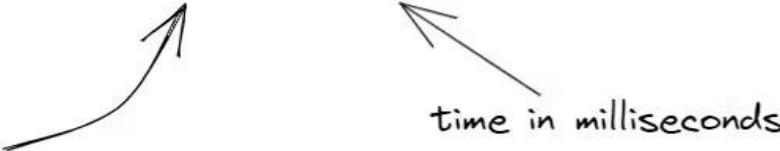
Pre-discussed: setInterval

`setInterval` is a JavaScript function that repeatedly executes a specified callback function at fixed time intervals (in milliseconds).

Syntax:

```
setInterval(greet,    time);  
  
function greet()  
{  
    console.log("hello")  
}
```

time in milliseconds



Pre-discussed: setInterval example

Here we are printing and updating message count every 1000 milliseconds. The `setInterval()` function returns a unique identifier, known as an interval ID, which is stored to later clear the interval

```
1 let count = 0;
2
3 const intervalId = setInterval(() => {
4     console.log(`Message ${++count}`);
5 }, 1000);
```

Output:

Message 1
Message 2
Message 3
..
..

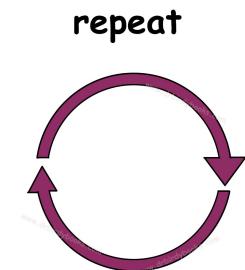
Usage: Setting up timers (SetInterval)

In similar fashion you can use setInterval if you want a task to be repeated after certain interval of time.

```
1  function repeatTask(ms) {  
2      return new Promise(() => {  
3          setInterval(() => {  
4              console.log("Repeating task...");  
5          }, ms);  
6      });  
7  }  
8  
9  console.log("Starting...");  
10  
11 repeatTask(2000);  
12  
13 console.log("Waiting...");
```

Creating and returning a promise

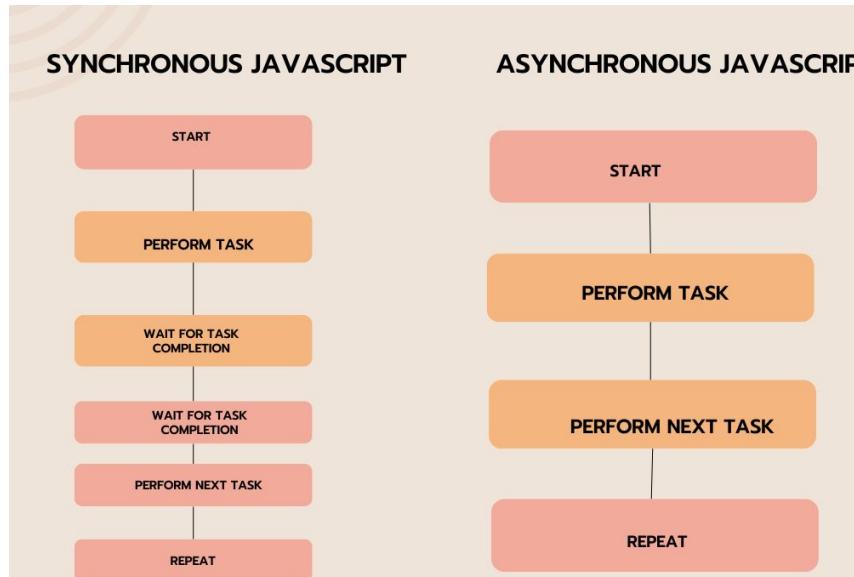
Calling the function returning the promise.



Understanding Asynchronous flow in Promises

What is Asynchronous Flow?

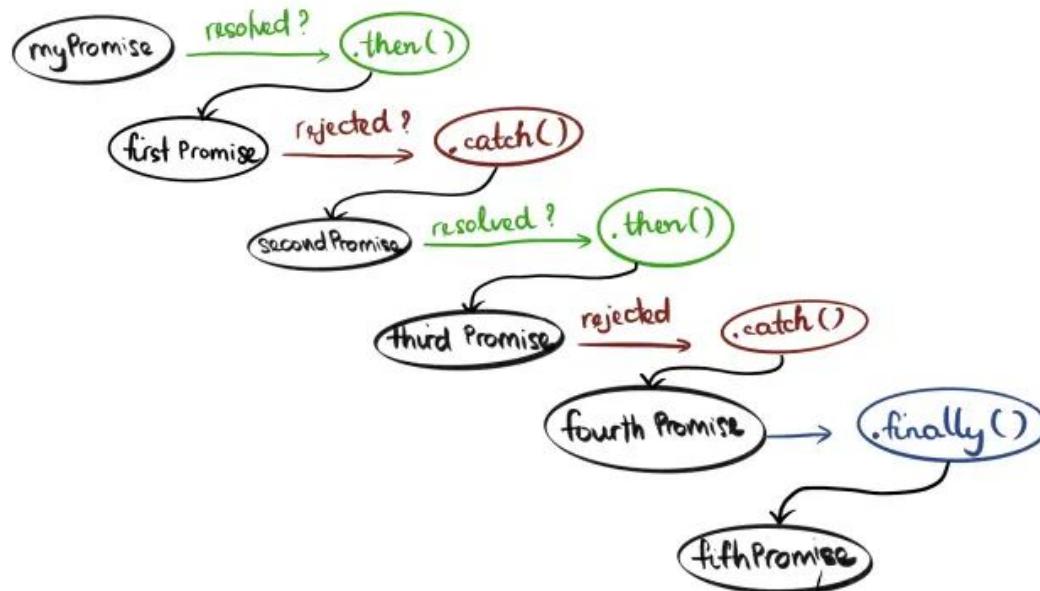
Asynchronous flow allows programs to perform tasks concurrently without blocking other tasks. It enables operations such as fetching data, reading files, or waiting for user input without freezing the program.



Unlike sync operations
async operations do not
need to wait till the other
task complete its
execution.

Asynchronous Flow in Promises

In promises we chain the tasks using then() and catch() blocks. Hence program is executed in chain like manner.

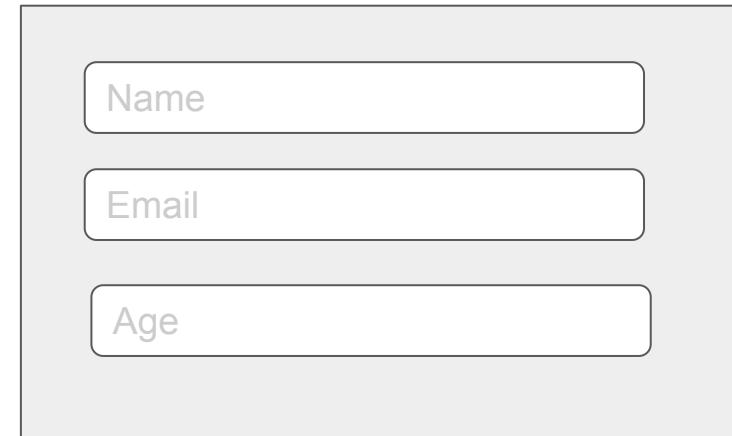
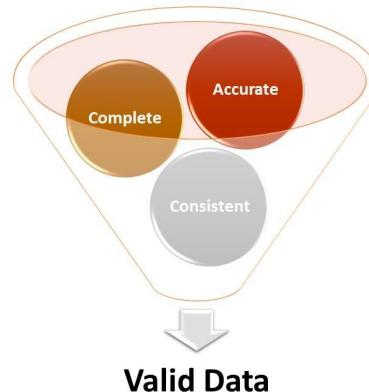


Corresponding block executes only after the execution of block previous block.

Workshop: Question

You're building a registration form where users submit their name, email, and age. The input data needs to be validated asynchronously, such as if name is valid, checking if the email is already registered (with a delay) and ensuring the age is valid.

How would you implement using promises?



A registration form consisting of three input fields:

- Name
- Email
- Age



Workshop: Solution

Let's write a function using promises for validating name asynchronously.



```
1 // Function to validate the name
2 function validateName(name) {
3     return new Promise((resolve, reject) => {
4         if (!name.trim()) {
5             reject('Name cannot be empty.');
6         } else if (name.length < 3) {
7             reject('Name must be at least 3 characters long.');
8         } else if (/[^a-zA-Z\s]/.test(name)) {
9             reject('Name must only contain letters and spaces.');
10        } else {
11            resolve('Name is valid.');
12        }
13    });
14 }
```

Write functions to validate email and age by yourself.

Workshop: Solution

Let's write validation functions for email and age too.



```
1 function validateEmail(email) {
2     return new Promise((resolve, reject) => {
3         const emailPattern =
4             /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/;
5         if (!emailPattern.test(email)) {
6             reject('Invalid email format.');
7         } else {
8             resolve('Email is valid.');
9         }
10    });
11 }
```

Validating Email



```
1 function validateAge(age) {
2     return new Promise((resolve, reject) => {
3         if (isNaN(age) || age < 18) {
4             reject('Must be a number and at least 18.');
5         } else {
6             resolve('Age is valid.');
7         }
8     });
9 }
```

Validating Age

In Class Questions

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/>

**Thanks
for
watching!**