Before we begin...

- Videos On!

Welcome

Agenda

- Asynchronous Programming
- Synchronous Programming
- Promises

Asynchronicity

Asynchronous vs Synchronous

- Programming languages are either *synchronous* or *asynchronous*
- Synchronous languages are where tasks are performed one at a time
  - It needs to wait for a task to finish before moving to the next one
- Asynchronous languages allow you to move on to another task before the previous one finishes

JavaScript is an asynchronous programming languages, which means we have to deal with asynchronicity (some things take time)

What takes time?

- Events
- User Interactions
- Animations
- Games
- API Calls

How do we deal with asynchronicity?

Callbacks

We have seen these!

`Promise`

We are about to see these

`async` and `await`

Very new features in JS, so we will see these later on

Promise

What are Promises?

- Let's think about what a Promise is, in general
- "A *Promise* is an object representing the eventual completion or failure of an **asynchronous operation**"
- It's an object that may produce a single piece of data at some point in the future
  - Either a resolved value
  - Or a rejection (an error that tells us why it wasn't resolved)
- A Promise can **only succeed or fail once**

States and Fates

A Promise can have one of three different *states* at any moment:

- **Fulfilled**: The action relating to the promise succeeded
- **Rejected**: The action relating to the promise failed
- **Pending**: The action hasn't been fulfilled or rejected yet

We say that a promise is **settled** if it isn't pending

States and Fates

A Promise can have one of two different *fates*:

- **Resolved**: Finished (or potentially moved into a *thenable* or another Promise)
- **Unresolved**: Unfinished (if resolving or rejecting will make an impact)

We say that a promise is **settled** if it isn't pending

Why use Promises?

- They help us write readable code
- They help us deal with the complexities of asynchronous programming
- They help us avoid "callback hell" or the "pyramid of doom"
- They are the backbone of some of the newer features coming out with JavaScript (e.g. `async` and `await`)
- Lots of libraries/frameworks/packages use Promises
  - So we will have to be **consumers of Promises**

Working with Promises

I will be showing you how to create Promises *but* we rarely have to do that

We will be **consuming Promises regularly**, so I just want to show you how it works very briefly behind the scenes

As always, *the concept is the most important part*

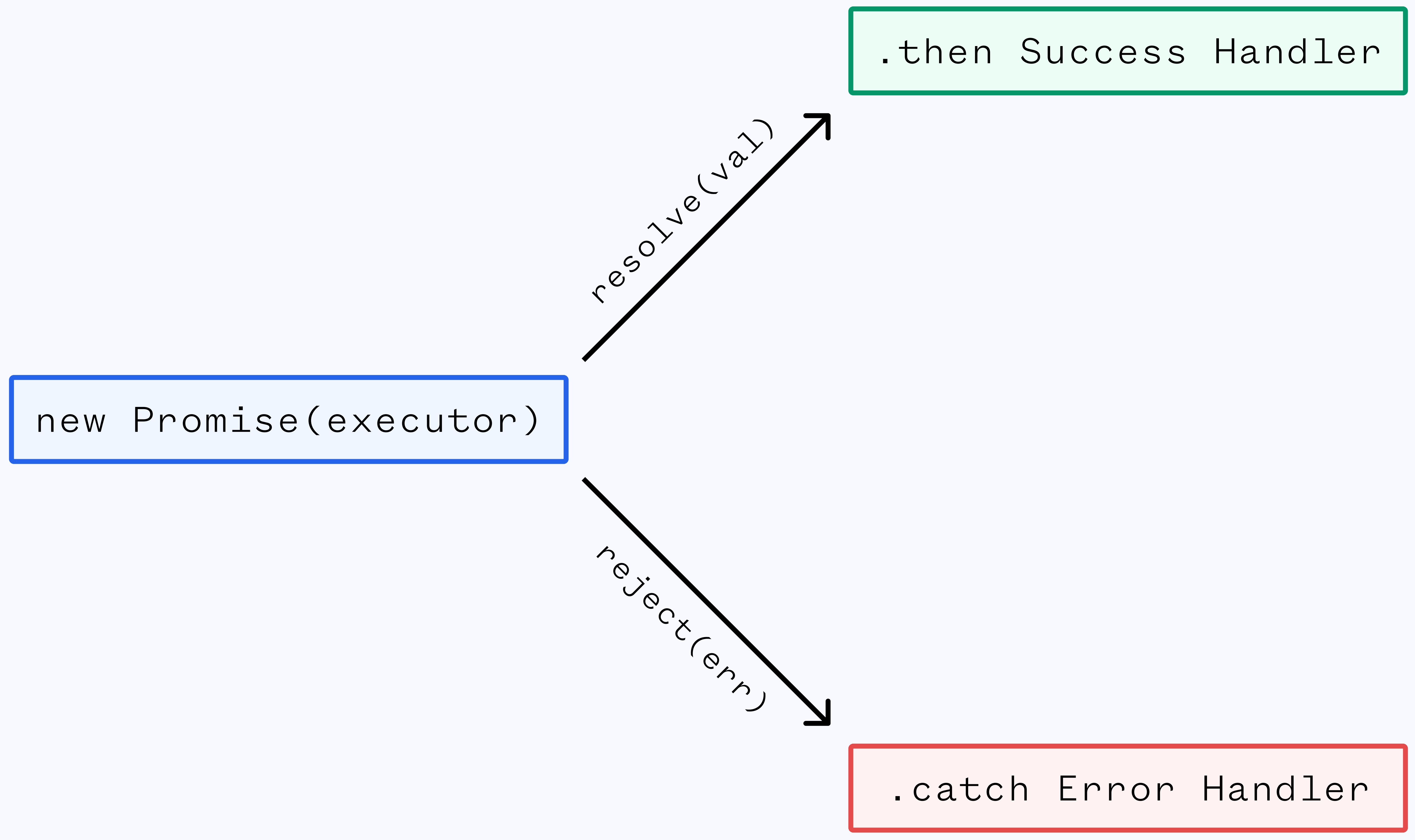
Some Terminology

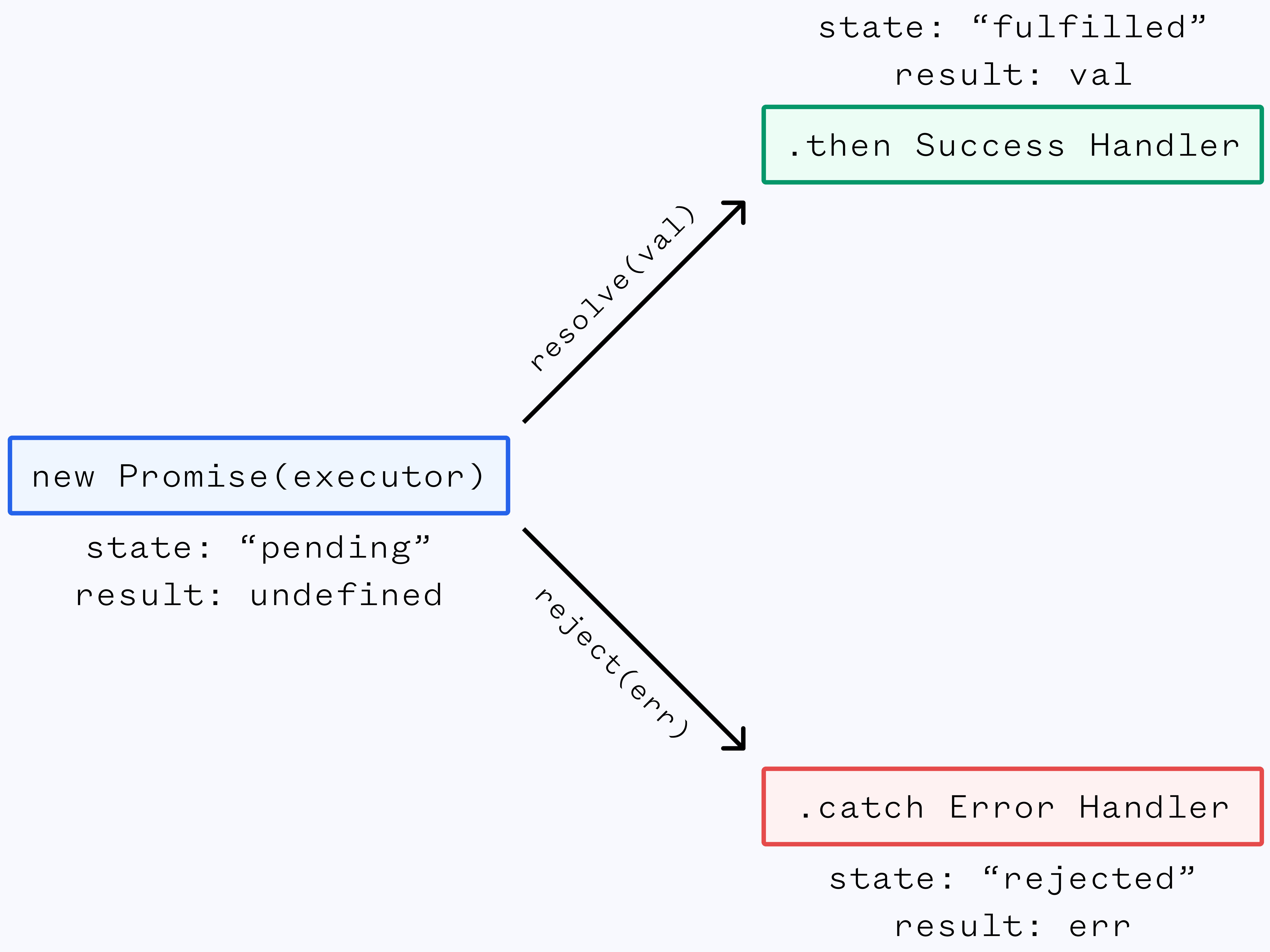
- **Executor**: A function that takes time (contains the producing code)
- **Fulfilled**: Succeeded
- **Rejected**: Failed
- **Pending**: Waiting
- **Settled**: Not waiting

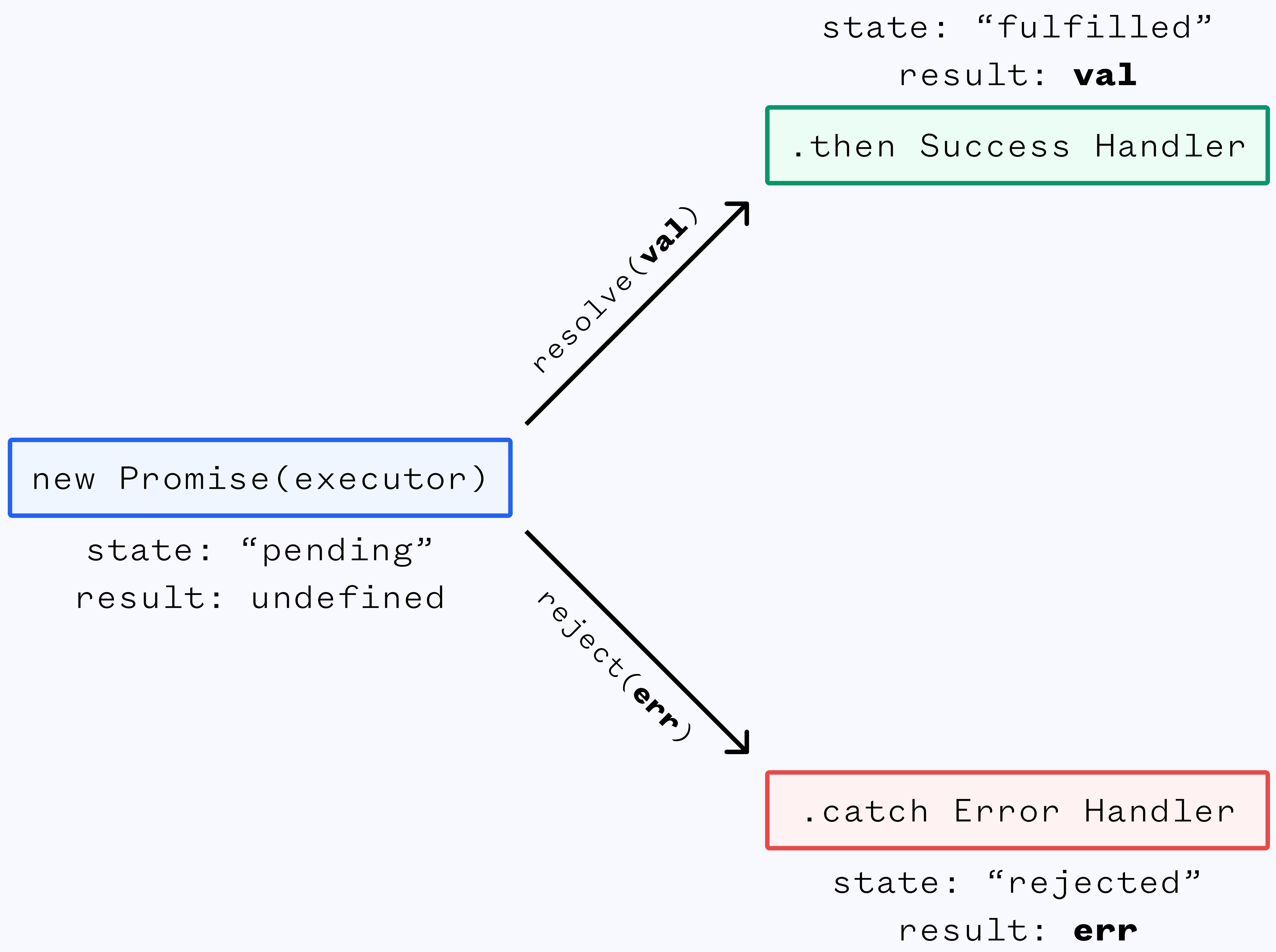
Some Terminology

- **Resolved**: Finished (whether successfully or unsuccessfully)
- **Unresolved**: Still work left to do (resolving or rejecting will make an impact)
- **Thenable**: A piece of data that is promise-like, and therefore has a `.then` method attached to it

How do they work?


new Promise(executor)
resolve(val)
.then Success Handler
reject(err)
.catch Error Handler


state: “fulfilled”
result: val
.then Success Handler
resolve(val)
new Promise(executor)
state: “pending”
result: undefined
reject(err)
.catch Error Handler
state: “rejected”
result: err


state: “fulfilled”
result: **val**
.then Success Handler
resolve(**val**)
new Promise(executor)
state: “pending”
result: undefined
reject(**err**)
.catch Error Handler
state: “rejected”
result: **err**

Creating Promises

Creating Promises

1. Create an executor function (just a regular function that should eventually be asynchronous - one that takes time)
2. Create a **new instance** of the **Promise** class and provide that executor function
3. That executor function will be run automatically and it will be provided with a **resolve** callback and a **reject** callback

Resolve

If we execute the **resolve** callback:

- It will change the state to "fulfilled" (mark it as successful)
- It will try to run the attached `.then` if it exists
- The `.then` callback function will be provided with whatever we passed to `resolve`

Reject

If we execute the **reject** callback:

- It will change the state to "rejected" (mark it as failed)
- It will try to run the attached `.catch` if it exists
- The `.catch` callback function will be provided with whatever we passed to `reject`

Creating Promises

```javascript
function myExecutor(resolve, reject) {
  resolve("Success!");
}

let myPromise = new Promise(myExecutor);
```

Creating Promises

```javascript
function successHandler(message) {
  console.log("Success", message);
}

function myExecutor(resolve, reject) {
  resolve("The file has been downloaded!");
}

let downloadFile = new Promise(myExecutor);

downloadFile.then(successHandler);
```

Creating Promises

```
function myExecutor(resolve, reject) {
  reject("Failure!");
}

let myPromise = new Promise(myExecutor);
```

Creating Promises

```
function errorHandler(message) {
  console.log("Failure", message);
}

function myExecutor(resolve, reject) {
  reject("You are unauthorized");
}

let signIn = new Promise(myExecutor);
signIn.catch(errorHandler);
```

Creating Promises

```javascript
function successHandler(message) {
  console.log("Success", message);
}

function errorHandler(message) {
  console.log("Failure", message);
}

function myExecutor(resolve, reject) {
  if (false) {
    resolve("Successfully logged in");
  } else {
    reject("Something went wrong");
  }
}

let signIn = new Promise(myExecutor);
signIn.then(successHandler).catch(errorHandler);
```

Exercise

Turn a timer into a promise (you'll have to use `setTimeout` within the executor callback)

I would like to be able to write the following code:

```
function afterASecond() {
  console.log("This has worked");
}

delay(1000).then(afterASecond);
```

Exercise

Turn an event into a promise (you'll have to use `addEventListener` somewhere)

I would like to be able to write the following code:

```
function onHeadingClick() {
  console.log("This has worked");
}

onFirstClick("h1").then(onHeadingClick);

// You'll have to remove the event listener too (use MDN)
```

Resources

- MDN: Promises and MDN: Using Promises
- Google Web Fundamentals: Promises
- JavaScript.info: Promises
- Scotch: JavaScript Promises
- David Walsh: Promises
- You Don't Know JS: Promises
- Exploring JS: Promises
- Eric Elliot: Promises
- Domenic: The Point of Promises

That's all for tonight!

Thank You!