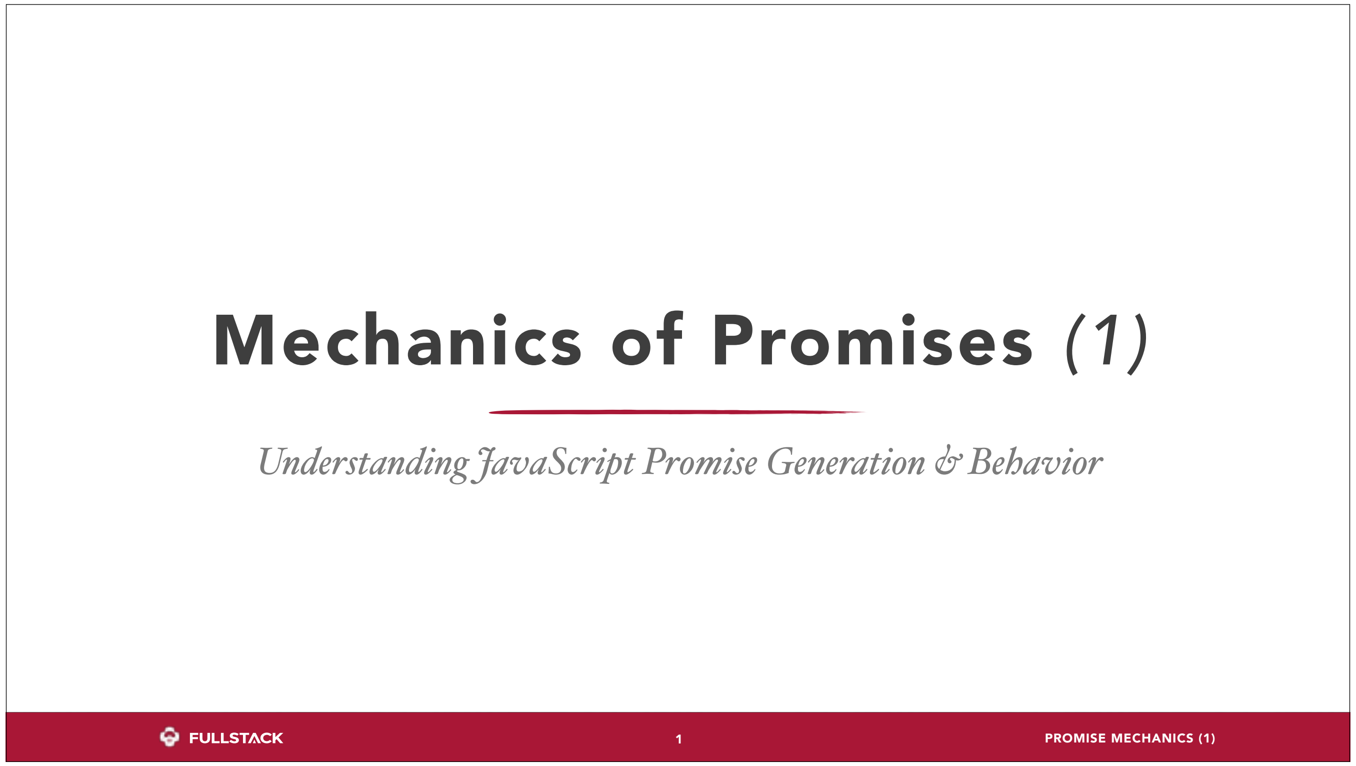


Mechanics of Promises *(1)*

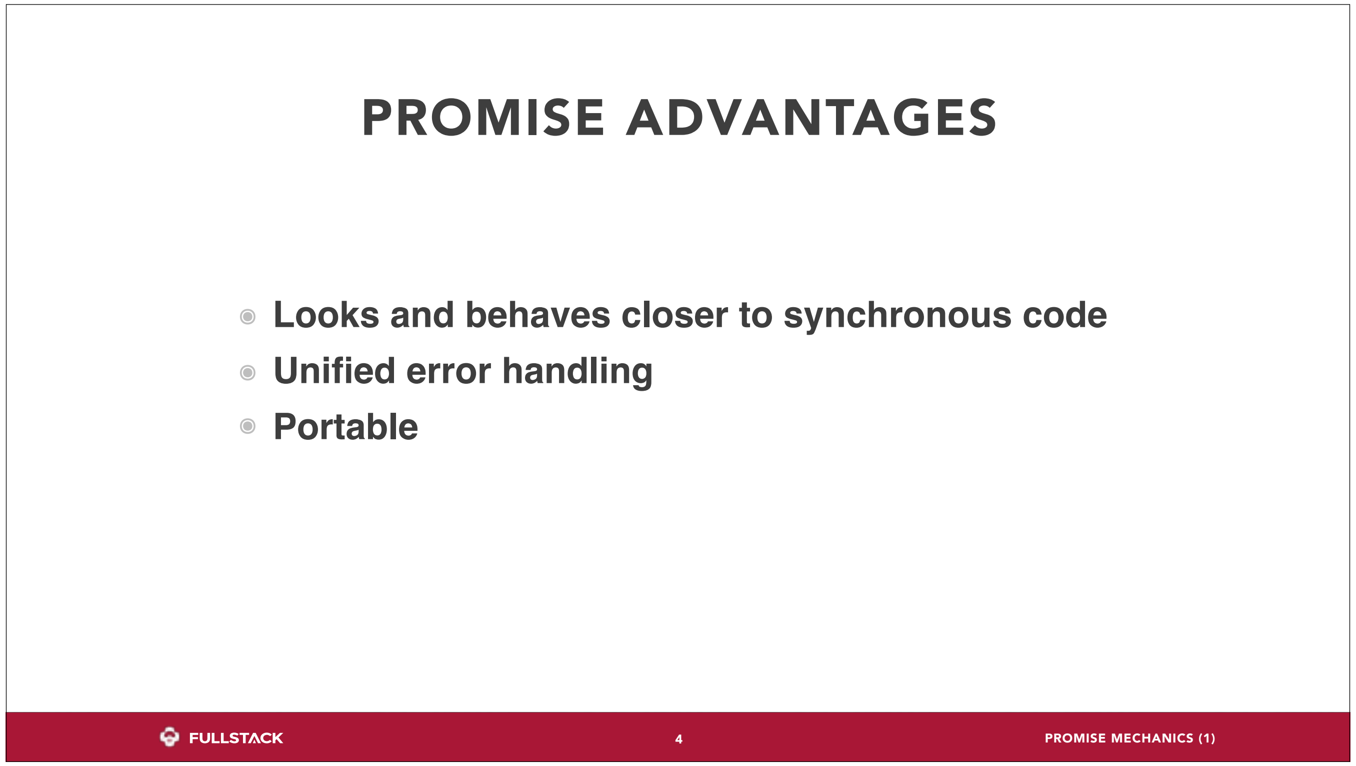
Understanding JavaScript Promise Generation & Behavior

We’ve already had a promise lecture…
We’ve had 1 yes, but what about second promise lecture?

Topics

- Why Promises
- Historical Context
- What is a Promise (in JS)
- Promise Global
- Creating new Promises

Why promises?



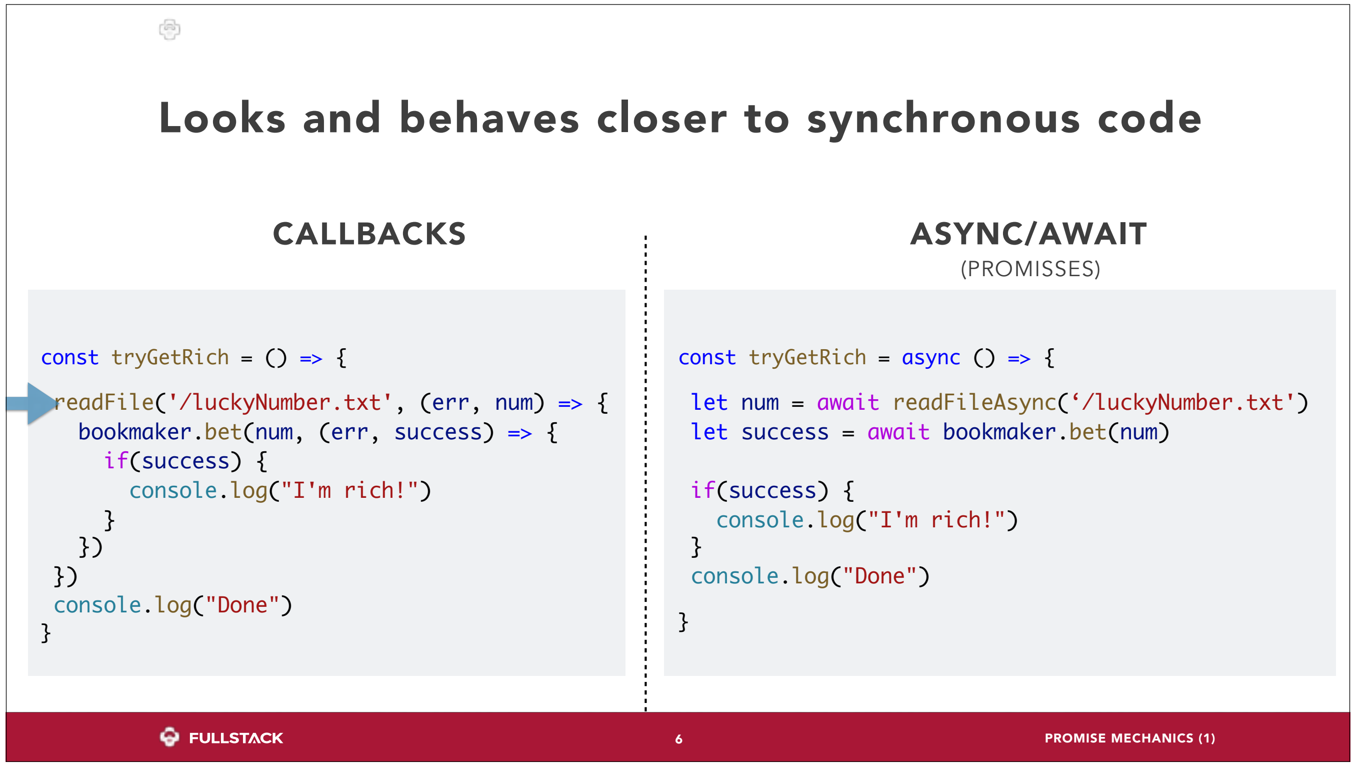
PROMISE ADVANTAGES

- Looks and behaves closer to synchronous code
- Unified error handling
- Portable

Though a bit harder to understand at first , there are great payoffs and promises end up making writing your code easier

PROMISE ADVANTAGES

- **Looks and behaves closer to synchronous code**
- Unified error handling
- Portable



Looks and behaves closer to synchronous code

CALLBACKS

```
const tryGetRich = () => {
 readFile('/luckyNumber.txt', (err, num) => {
   bookmaker.bet(num, (err, success) => {
     if(success) {
       console.log("I'm rich!")
     }
   })
 })
 console.log("Done")
}
```

ASYNC/AWAIT

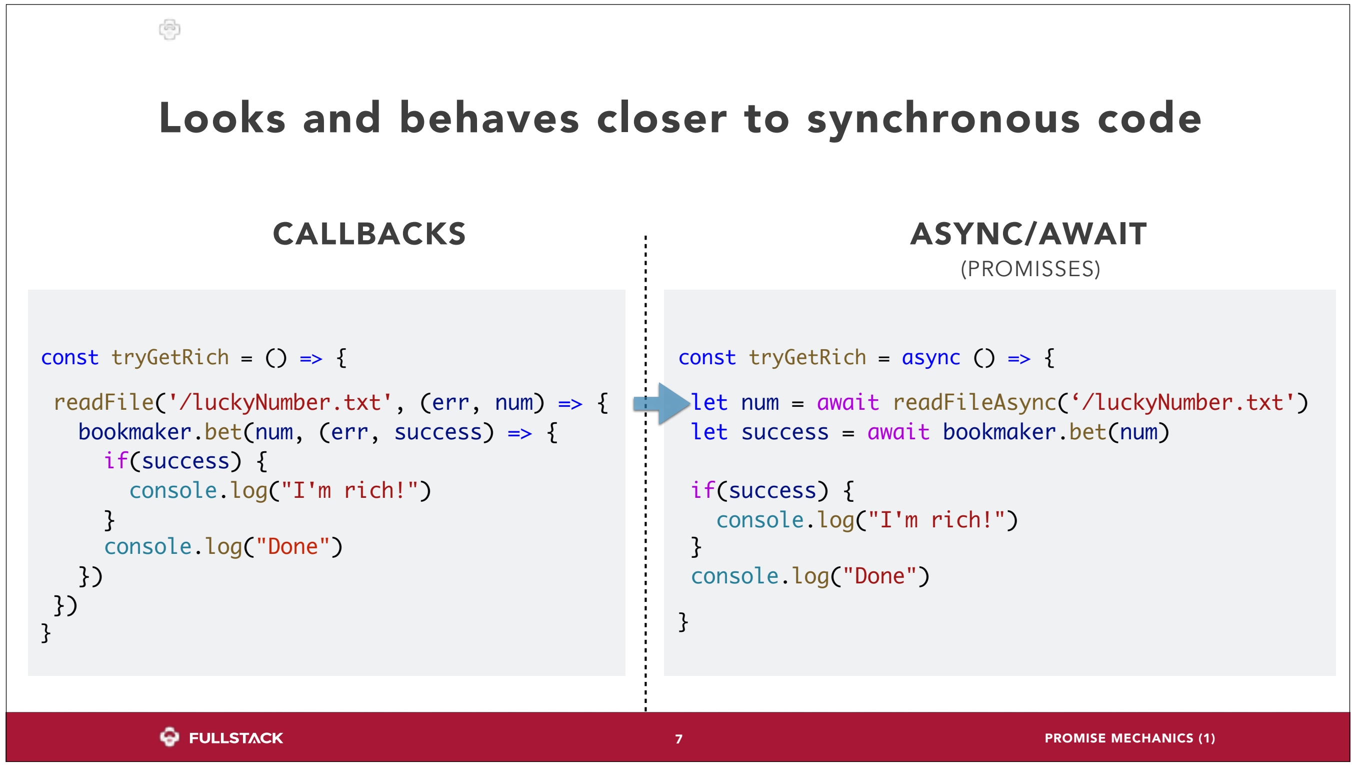
(PROMISSES)

```
const tryGetRich = async () => {
 let num = await readFileAsync(‘/luckyNumber.txt')
 let success = await bookmaker.bet(num)

 if(success) {
   console.log("I'm rich!")
 }
 console.log("Done")
}
```

Before we get into more details, let’s remember the fundamental difference between providing a callback function or awaiting a promise.

For one, the code is more linear, you don’t have that pyramid of doom on the that can happen with multiple callbacks.

But also, with callback based async, code, things will not be executed in the same order they were written.



Looks and behaves closer to synchronous code

CALLBACKS

```
const tryGetRich = () => {

 readFile('/luckyNumber.txt', (err, num) => {
   bookmaker.bet(num, (err, success) => {
     if(success) {
       console.log("I'm rich!")
     }
     console.log("Done")
   })
 })
}
```

ASYNC/AWAIT

(PROMISSES)

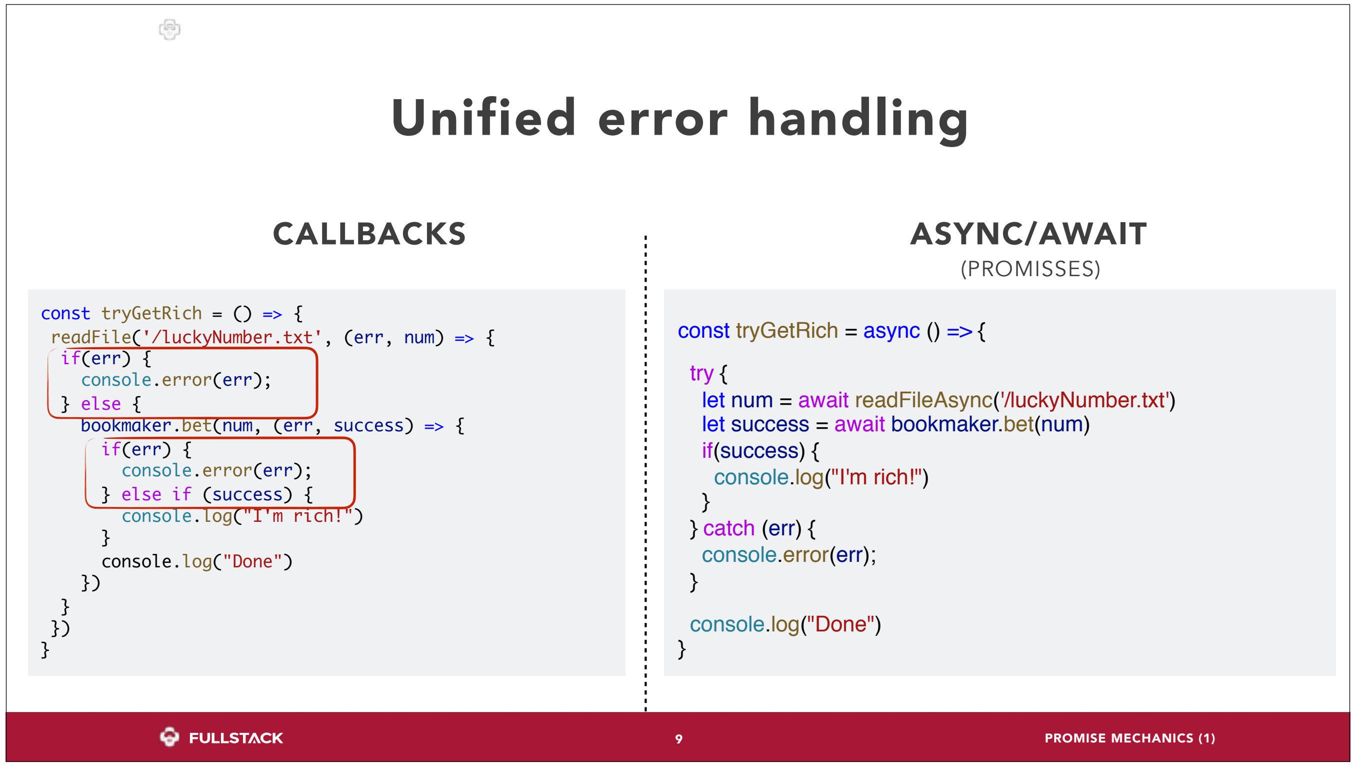
```
const tryGetRich = async () => {

 let num = await readFileAsync(‘/luckyNumber.txt')
 let success = await bookmaker.bet(num)

 if(success) {
   console.log("I'm rich!")
 }
 console.log("Done")

}
```

By using promises and async/await, your code look and behaves like synchronous code.

But there are another advantages to promises that we didn’t even talk about…

PROMISE ADVANTAGES

- Looks and behaves closer to synchronous code
- Unified error handling
- Portable



Unified error handling

CALLBACKS

```
const tryGetRich = () => {
 readFile('/luckyNumber.txt', (err, num) => {
  if(err) {
    console.error(err);
  } else {
    bookmaker.bet(num, (err, success) => {
      if(err) {
        console.error(err);
      } else if (success) {
        console.log("I'm rich!")
      }
      console.log("Done")
    })
  }
 })
}
```

ASYNC/AWAIT

(PROMISSES)

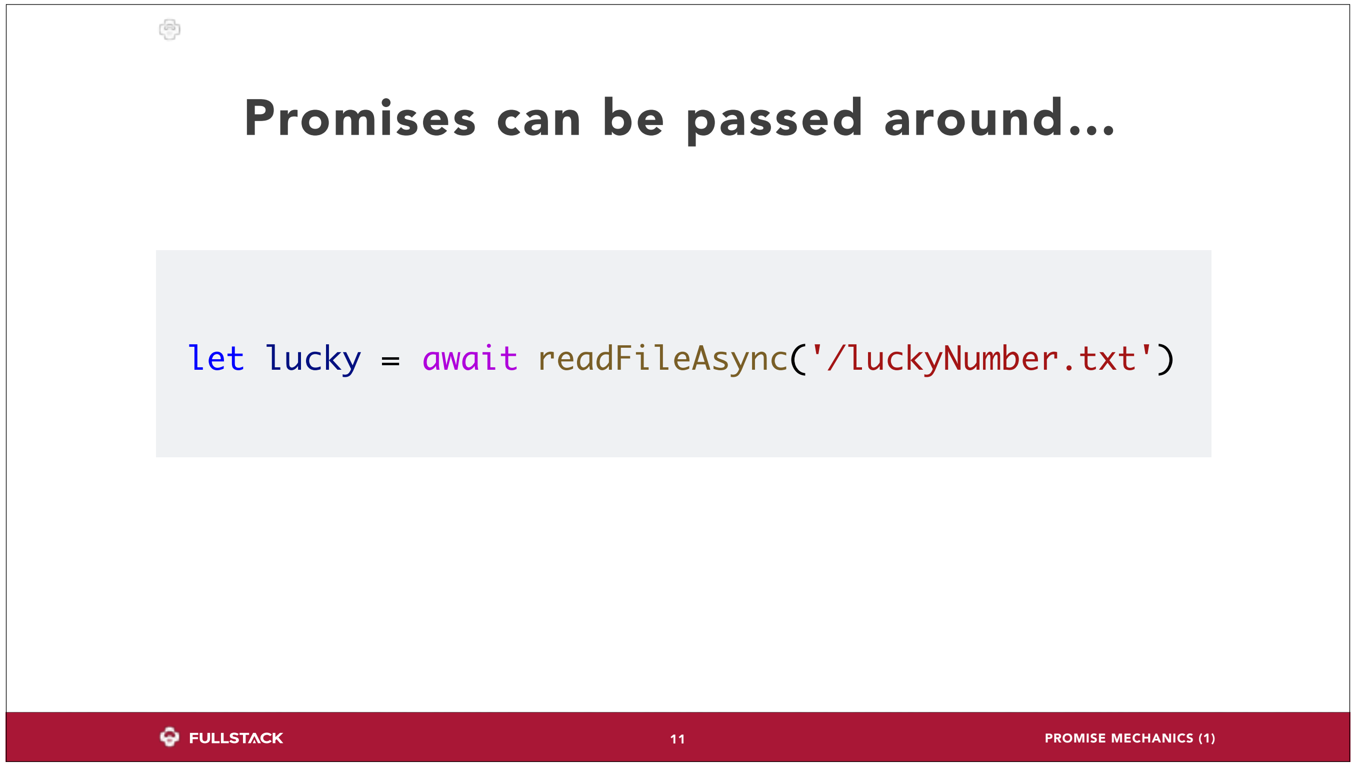
```
const tryGetRich = async () => {

  try {
    let num = await readFileAsync('/luckyNumber.txt')
    let success = await bookmaker.bet(num)
    if(success) {
      console.log("I'm rich!")
    }
  } catch (err) {
    console.error(err);
  }

  console.log("Done")
}
```

When using traditional callbacks to control our async code, we must handle the error inside of every callback. Which gets very annoying and long very quickly.

With promises, we only have to handle it in one place, and any error thrown within the try block will get caught.

PROMISE ADVANTAGES

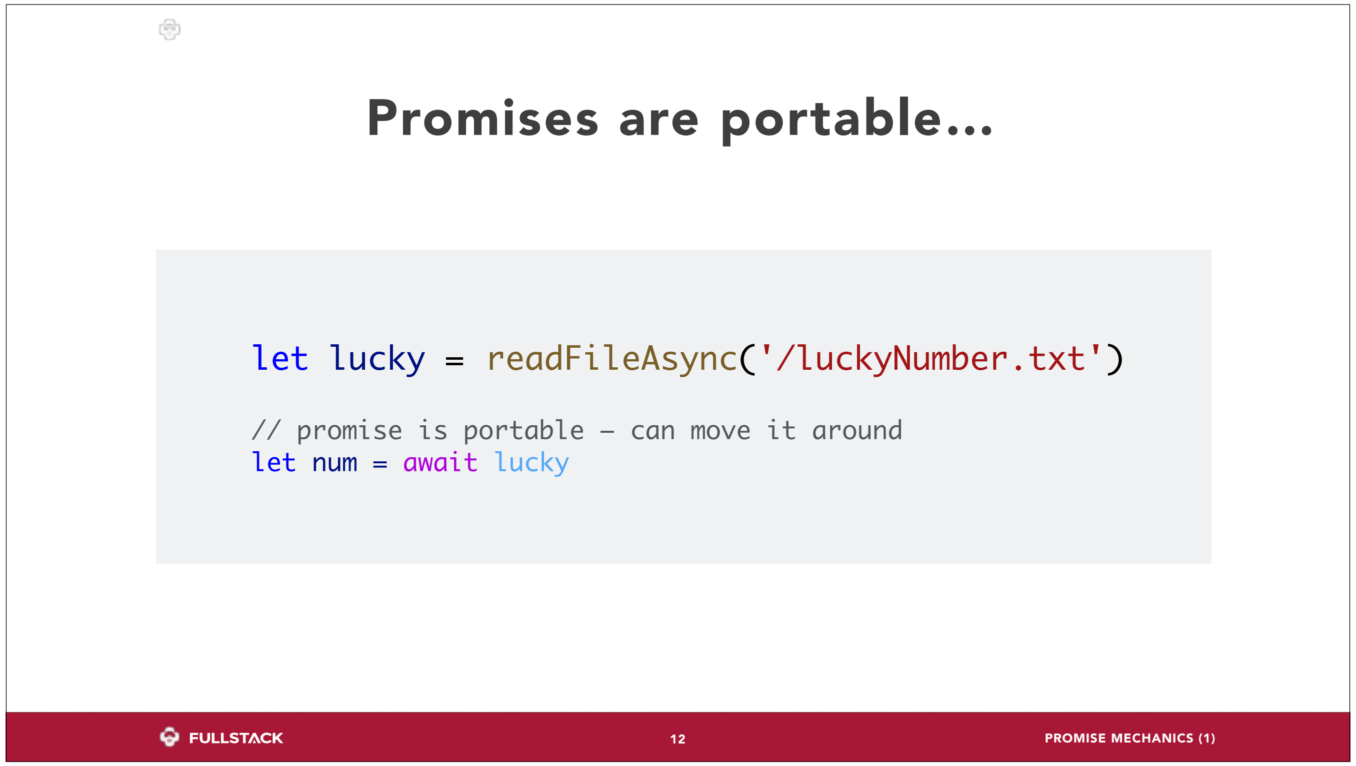
- Looks and behaves closer to synchronous code
- Unified error handling
- **Portable**



Promises can be passed around…

```
let lucky = await readFileAsync('/luckyNumber.txt')
```

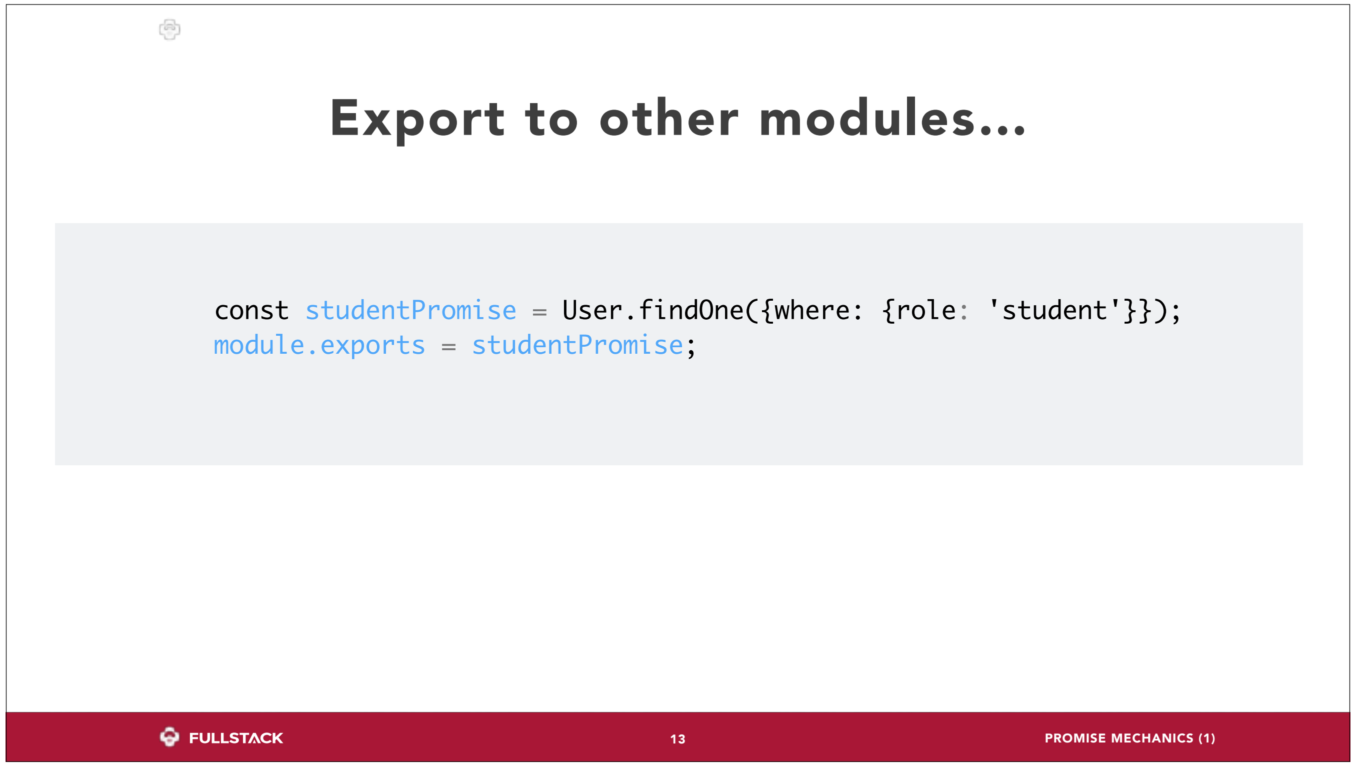
For example, what happens if I don’t `await` on a promise?
instead of waiting for a promise to resolve, you can get a reference to the promise object itself.



Promises are portable…

```
let lucky = readFileAsync('/luckyNumber.txt')

// promise is portable - can move it around
let num = await lucky
```

This means that promises are portable, they can be passed around….
Wherever the promise it passed, I can await the value.
I can even copy the promise into another variable



Export to other modules…

```
const studentPromise = User.findOne({where: {role: 'student'}});
module.exports = studentPromise;
```

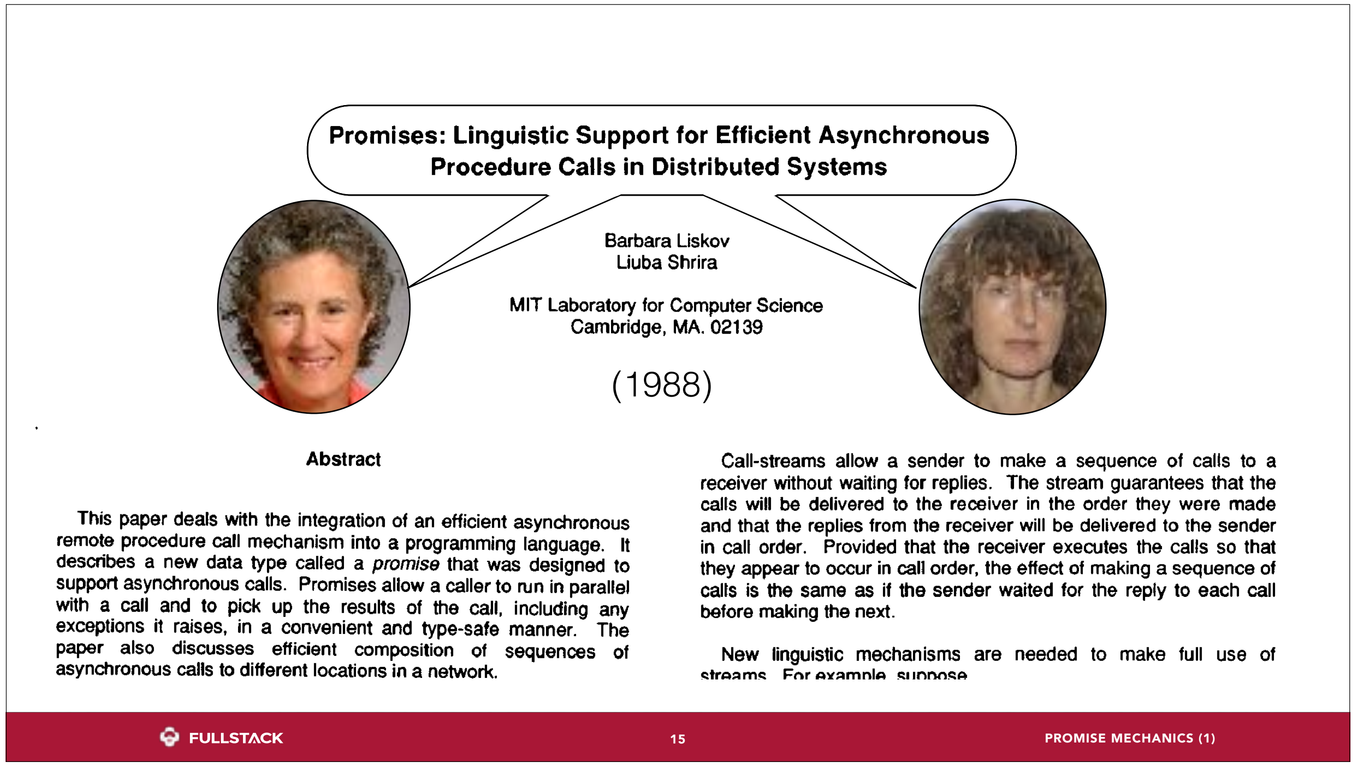
This portability lets us do a lot and can have big impact in the way we architecture applications.

Show portability via code example

We discuss this aspect more in part two of this lecture, but for now let’s take a step back…

How Did We End Up Here?

If promises are superior to callbacks, why some things in JS still doesn’t return promises - like Node’s built in modules? To understand that, let’s take a look on the history of promises.



Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems

Barbara Liskov
Liuba Shrira

MIT Laboratory for Computer Science
Cambridge, MA. 02139

(1988)

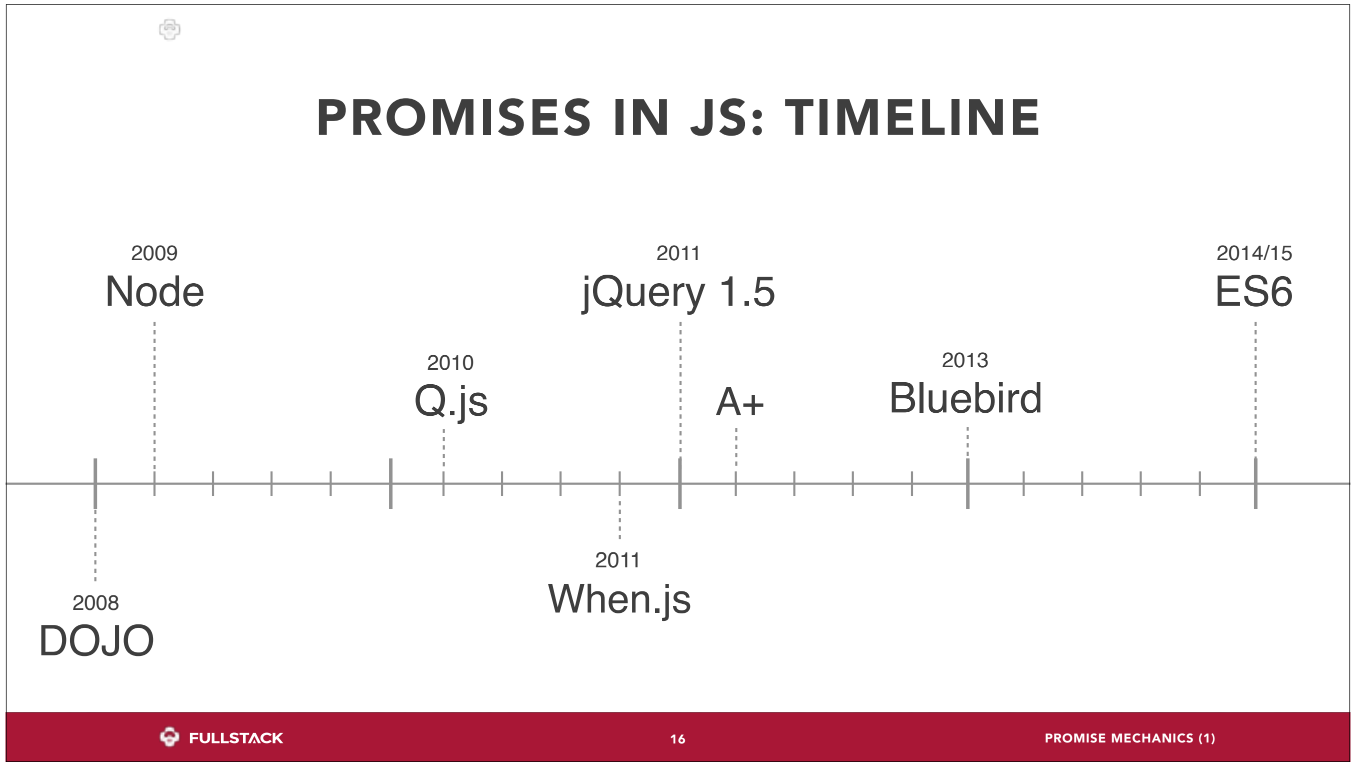
Abstract

This paper deals with the integration of an efficient asynchronous remote procedure call mechanism into a programming language. It describes a new data type called a *promise* that was designed to support asynchronous calls. Promises allow a caller to run in parallel with a call and to pick up the results of the call, including any exceptions it raises, in a convenient and type-safe manner. The paper also discusses efficient composition of sequences of asynchronous calls to different locations in a network.

Call-streams allow a sender to make a sequence of calls to a receiver without waiting for replies. The stream guarantees that the calls will be delivered to the receiver in the order they were made and that the replies from the receiver will be delivered to the sender in call order. Provided that the receiver executes the calls so that they appear to occur in call order, the effect of making a sequence of calls is the same as if the sender waited for the reply to each call before making the next.

New linguistic mechanisms are needed to make full use of streams. For example, suppose

The history of Promise starts a long time ago, in early 1980’s; first implementations began to appear in languages such as Prolog and Lisp as early as the 1980’s. The word “Promise” was coined by Barbara Liskov and Liuba Shrira in an academic paper called “Promises: linguistic support for efficient asynchronous procedure calls in distributed systems” (1988).



PROMISES IN JS: TIMELINE

2008
DOJO

2009
Node

2010
Q.js

2011
When.js

2011
jQuery 1.5

A+

2013
Bluebird

2014/15
ES6

This timeline helps explains why Node's built in modules doesn't return a promise: When Node was introduced, error-first callbacks was the settled standard for handle asynchronous behavior.

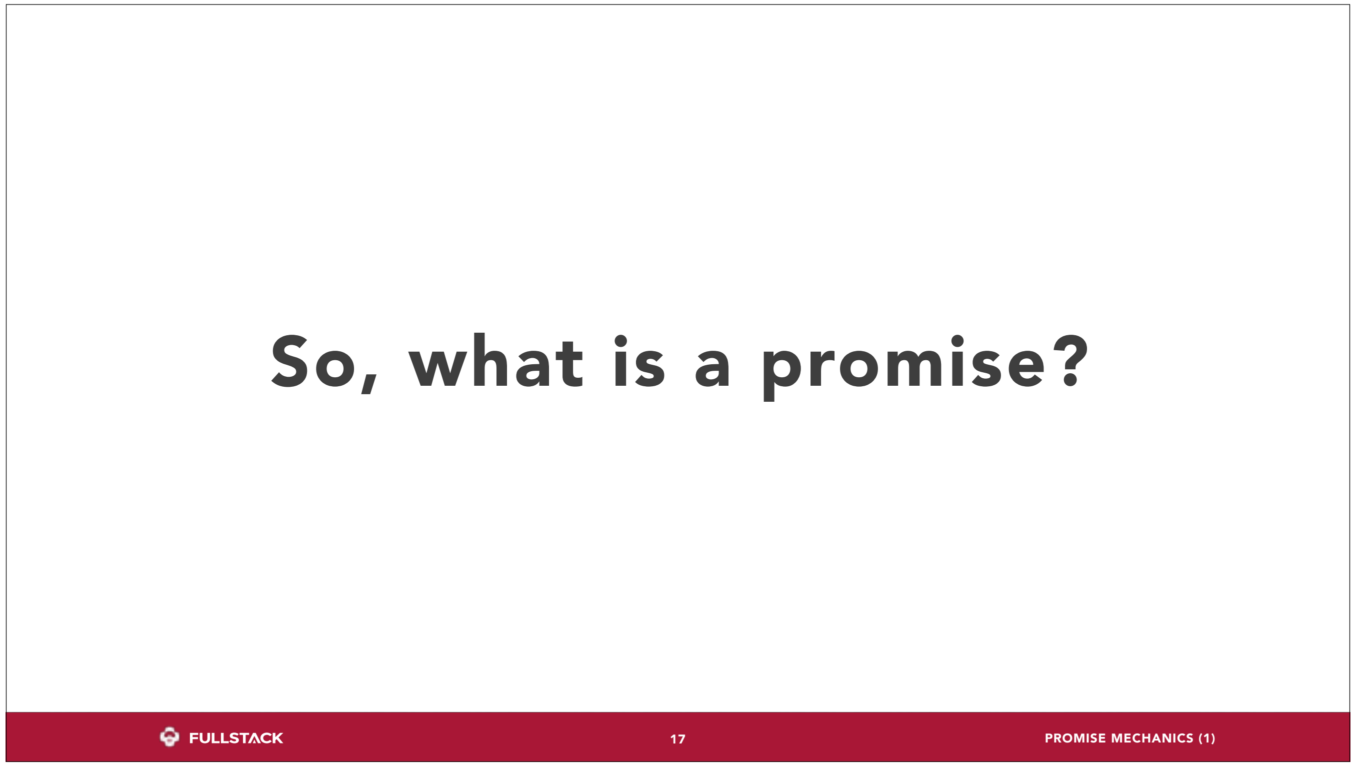
The community were looking for better alternatives - around the same time, the Dojo toolkit (a front-end library for the browser) added promises via the Deferred API. Growing interest in Promises led to a number of other promise libraries for JS (Q, When, RSVP, Bluebird…)

In 2011 jQuery's 1.5 adds promises. Due to jQuery's immense popularity, promises becomes mainstream.

jQuery's implementation incompatibilities motivated some important clarifications in the Promise spec, which was rewritten and rebranded as the Promises/A+ specification.

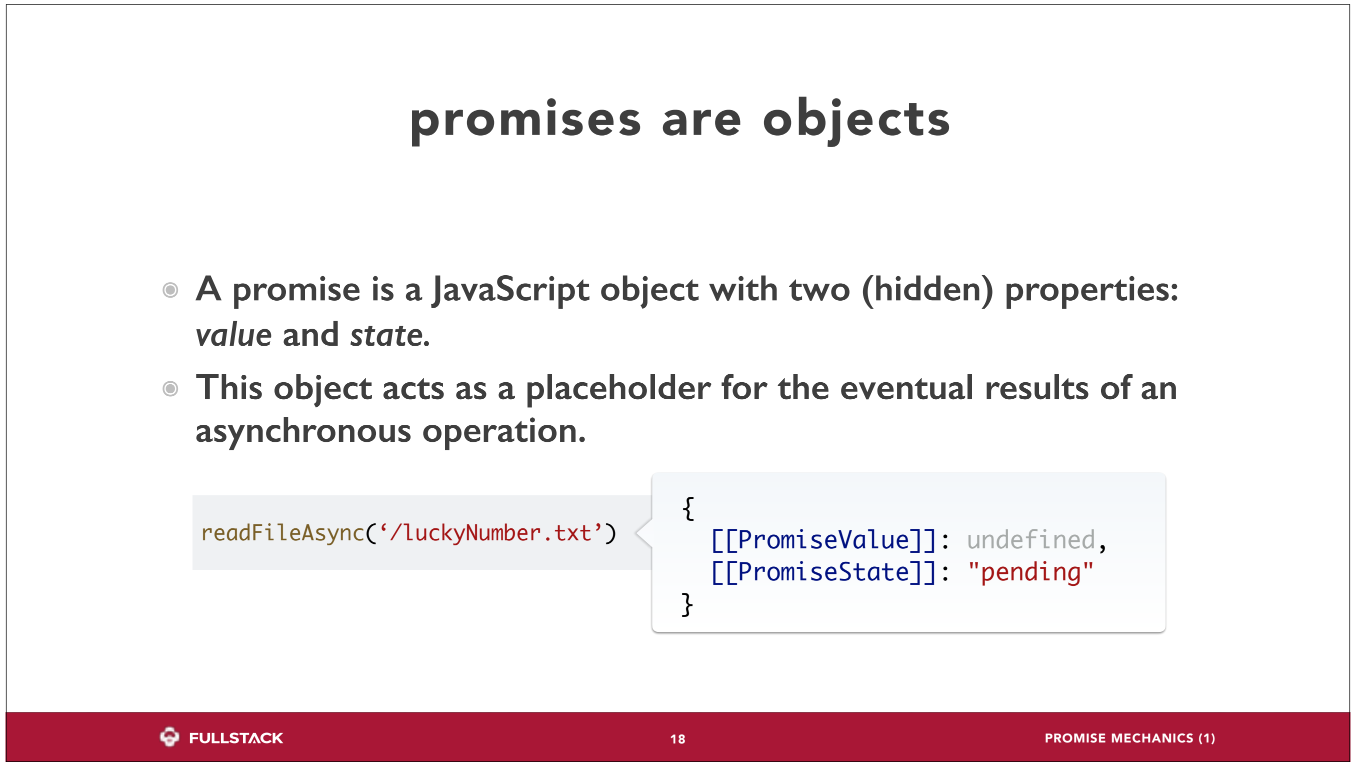
Also in 2011, the Promises/A specification was designed to make various promises more interoperable.

In 2014, ES6 brought a Promises/A+ compliant Promise global as a standard language feature.



So, what is a promise?

Story time. There was a Mother and her daughter living near a lake….



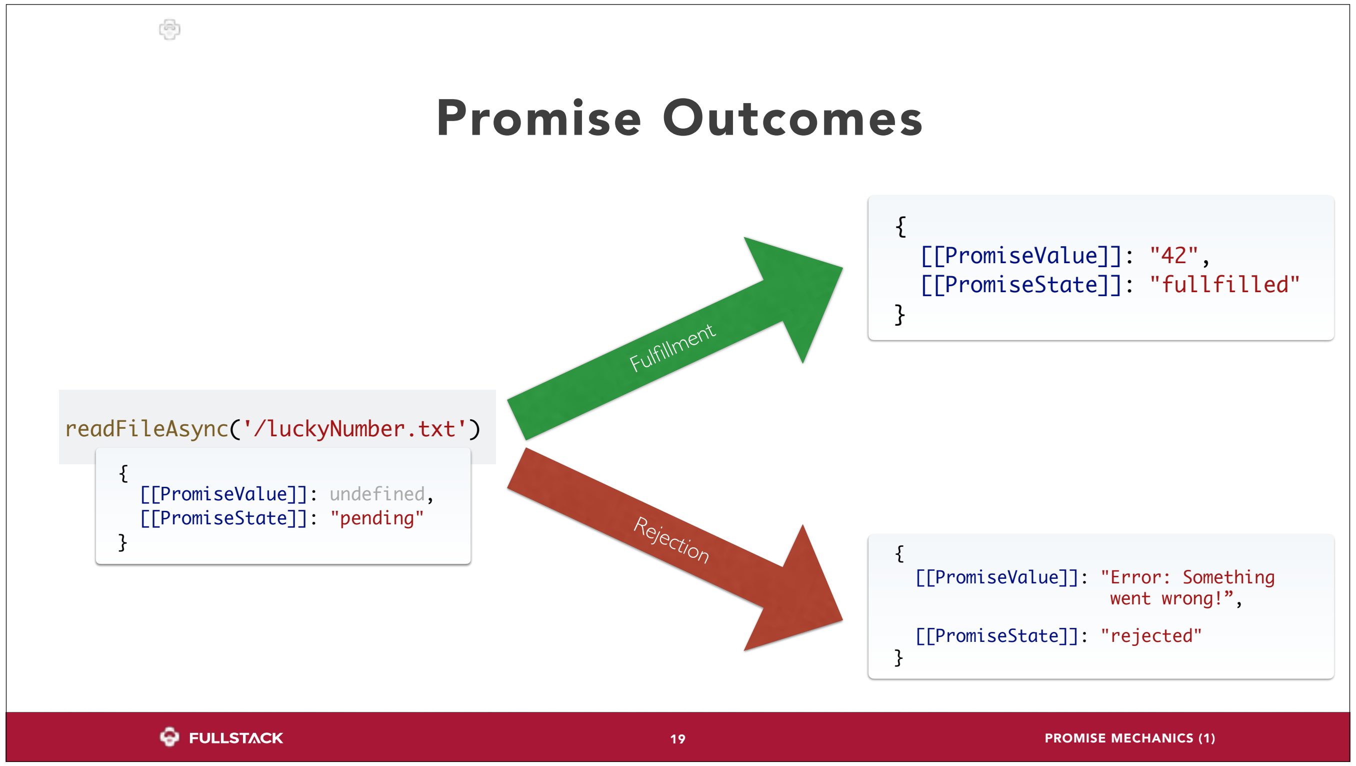
promises are objects

- A promise is a JavaScript object with two (hidden) properties: *value* and *state.*
- This object acts as a placeholder for the eventual results of an asynchronous operation.

```
readFileAsync(‘/luckyNumber.txt’)
```

```
{
  [[PromiseValue]]: undefined,
  [[PromiseState]]: "pending"
}
```

Promise objects encapsulate two crucial variables: Status and value;



Promise Outcomes

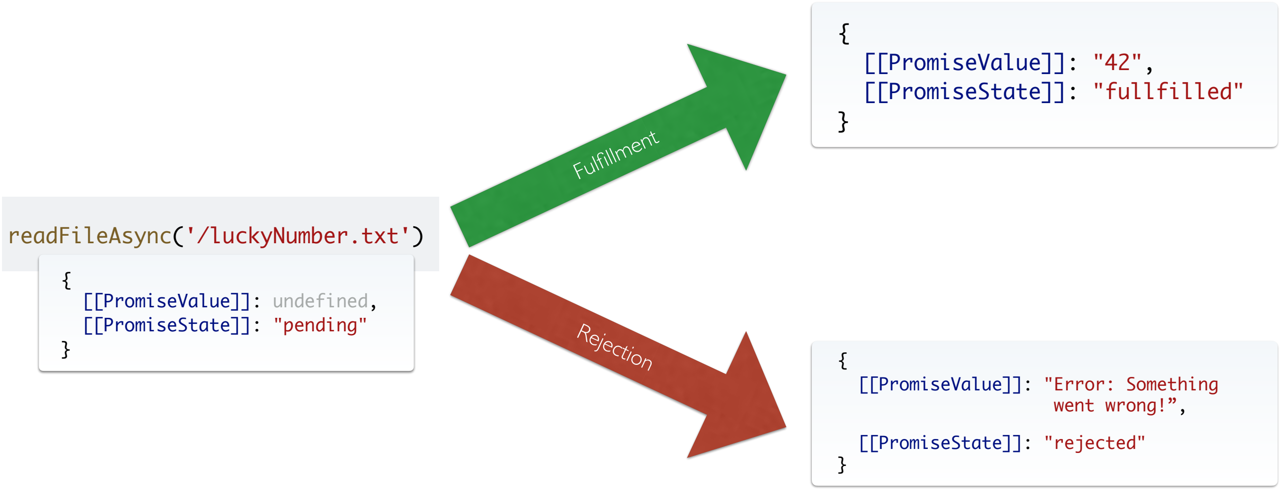
```
readFileAsync('/luckyNumber.txt')
{
  [[PromiseValue]]: undefined,
  [[PromiseState]]: "pending"
}
```

```
{
  [[PromiseValue]]: "42",
  [[PromiseState]]: "fullfilled"
}
```

```
{
  [[PromiseValue]]: "Error: Something
                     went wrong!”,

  [[PromiseState]]: "rejected"
}
```

The status starts as “pending” and ends up as “fulfilled" or “rejected”.
The eventual value will end up being one of the following:
- a fulfilled value (the data you were expecting for)
- a rejection reason (ie. a networking error)

Promise Outcomes

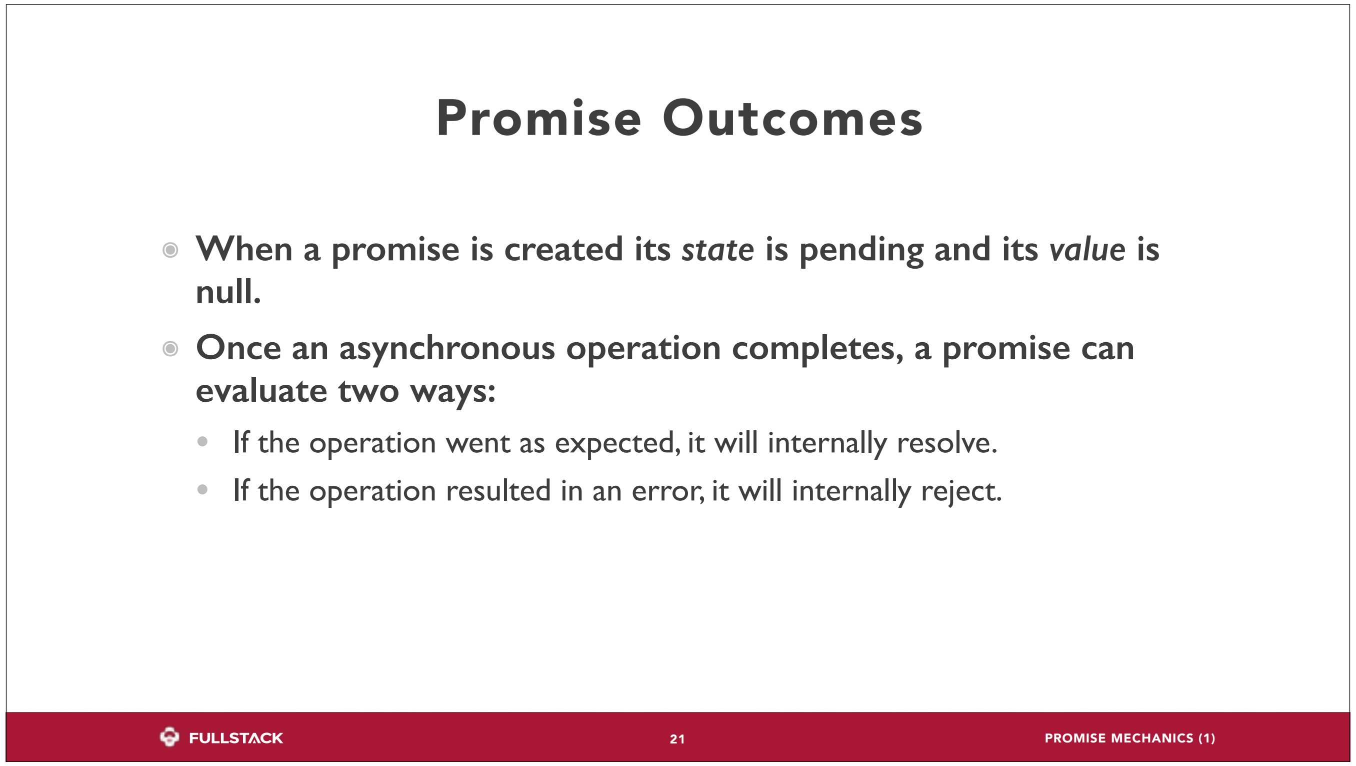


```
readFileAsync('/luckyNumber.txt')
```

```
{
  [[PromiseValue]]: undefined,
  [[PromiseState]]: "pending"
}
```

```
{
  [[PromiseValue]]: "42",
  [[PromiseState]]: "fullfilled"
}
```

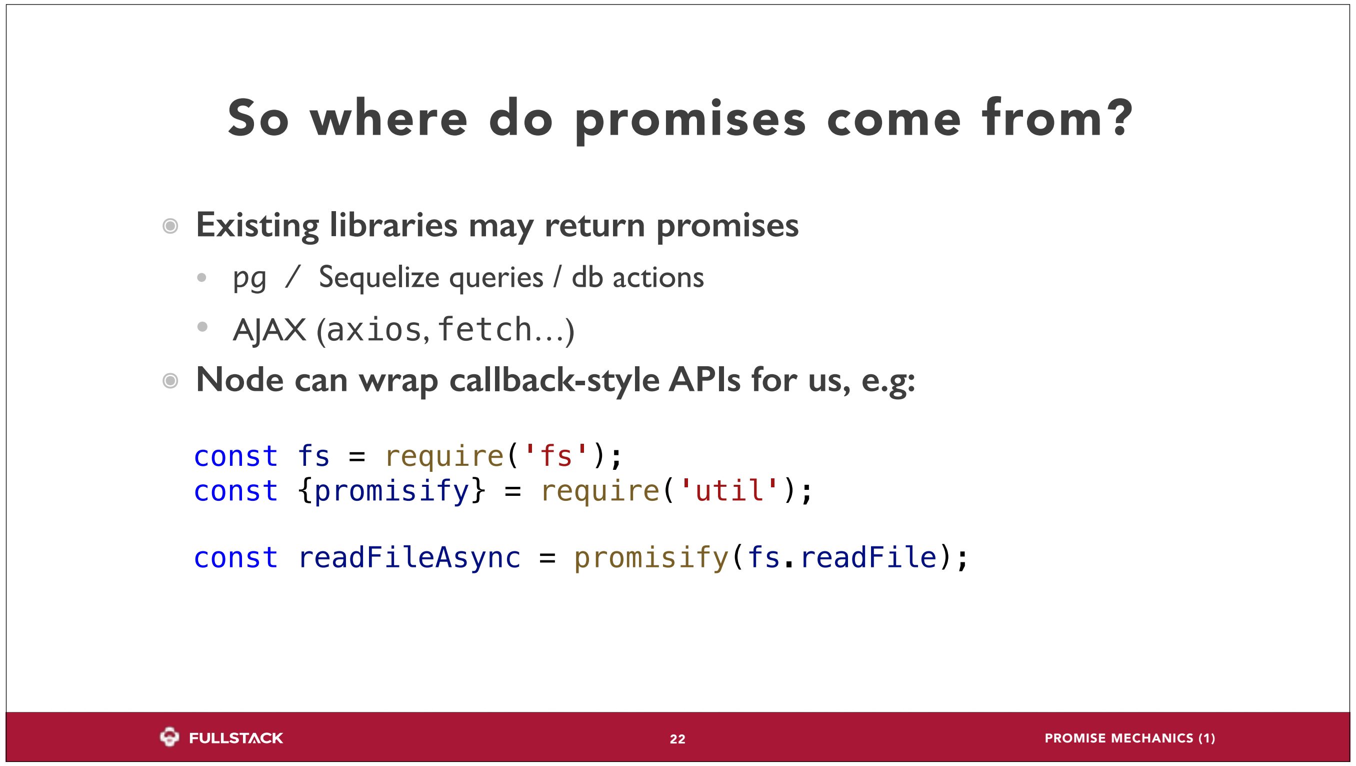
```
{
  [[PromiseValue]]: "Error: Something
                     went wrong!”,

  [[PromiseState]]: "rejected"
}
```



Promise Outcomes

- **When a promise is created its *state* is pending and its *value* is null.**
- **Once an asynchronous operation completes, a promise can evaluate two ways:**
  - If the operation went as expected, it will internally resolve.
  - If the operation resulted in an error, it will internally reject.

Well, there are a few things other than value and status, but we'll talk about that later.



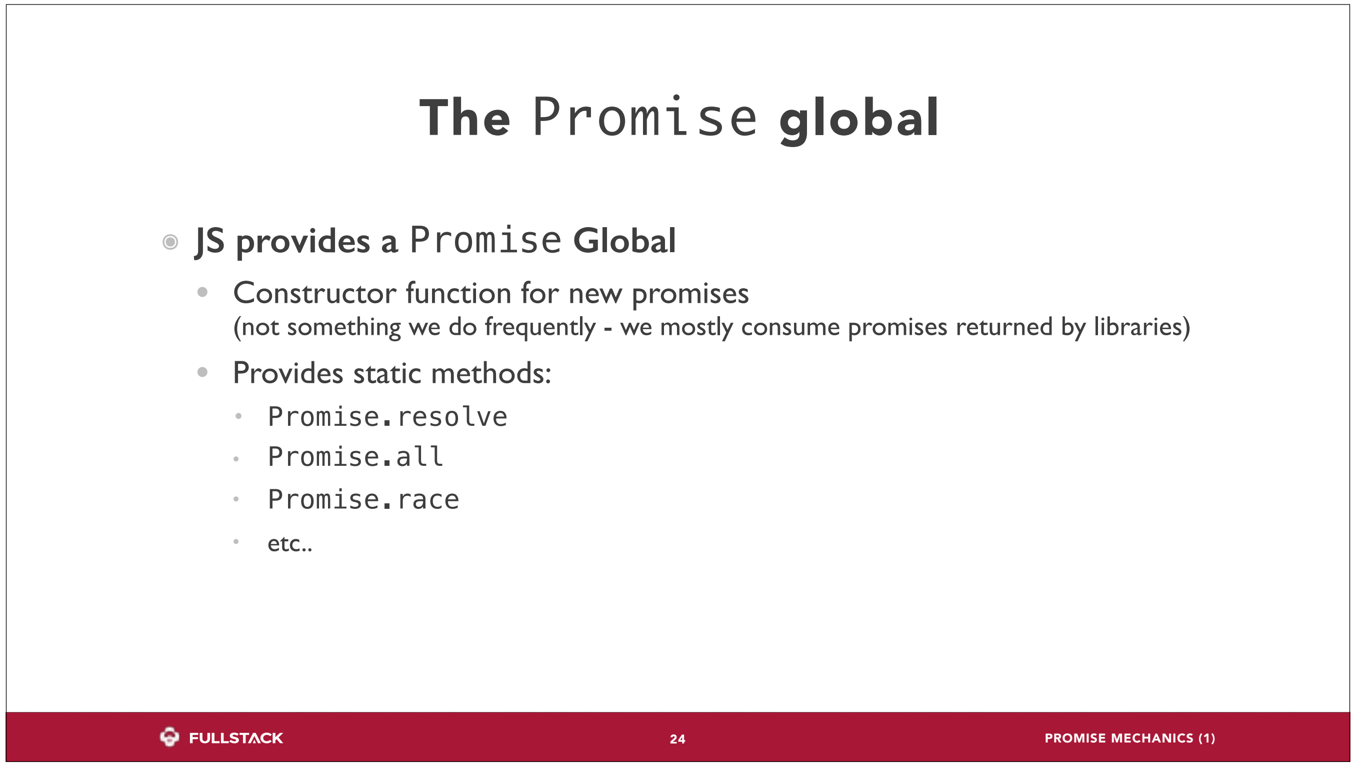
So where do promises come from?

- **Existing libraries may return promises**
  - `pg` / Sequelize queries / db actions
  - AJAX (`axios`, `fetch`…)
- **Node can wrap callback-style APIs for us, e.g:**

```
const fs = require('fs');
const {promisify} = require('util');

const readFileAsync = promisify(fs.readFile);
```

So far we’ve been consuming promise objects.
But how do we get a promise? Well, obviously some libraries return them for you. For node built-in modules, we can wrap them in promises using `utils.promisify`.

Making New Promises: How?

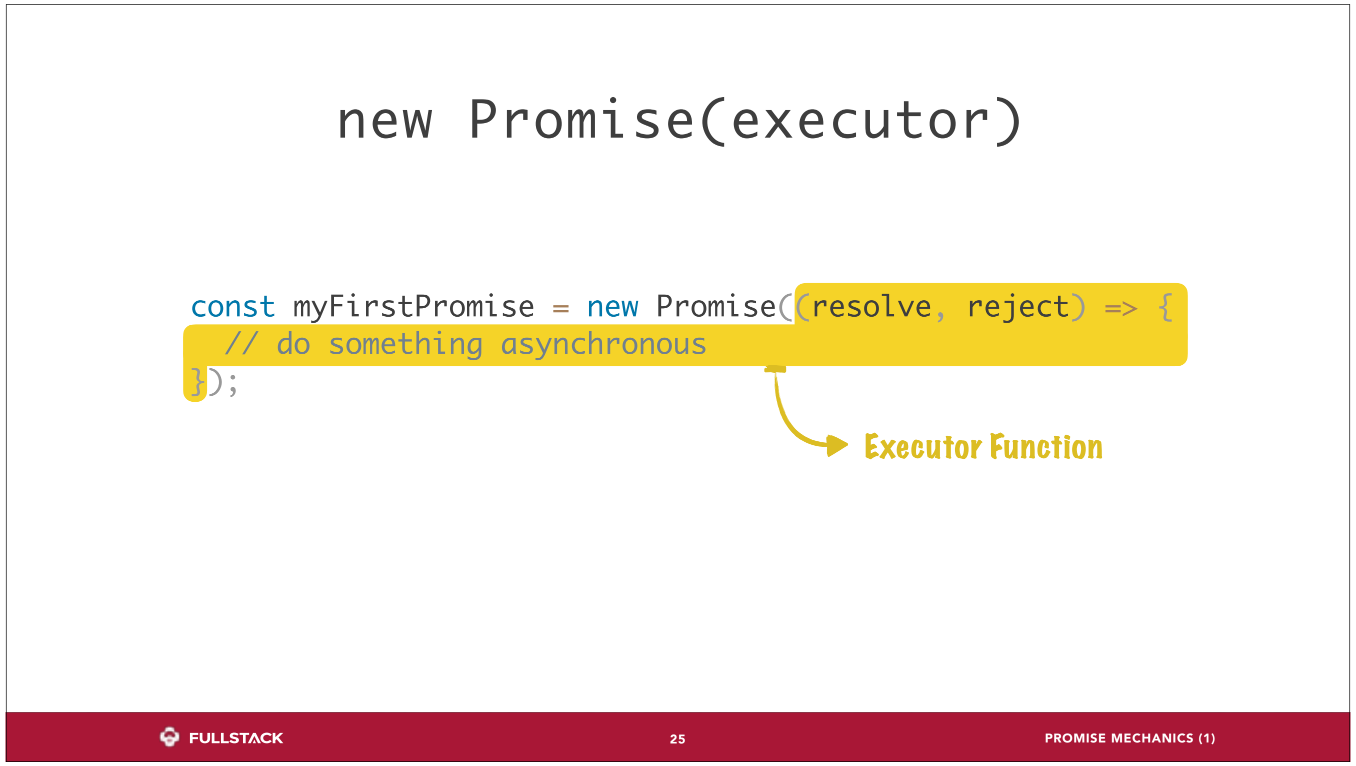


The `Promise` global

- JS provides a `Promise` Global
  - Constructor function for new promises
    (not something we do frequently - we mostly consume promises returned by libraries)
  - Provides static methods:
    - `Promise.resolve`
    - `Promise.all`
    - `Promise.race`
    - etc..

Creating new promises is not something you will do frequently - you will mostly consume promises returned by libraries such as sequelize. But if you’re dealing with lower level http operations, or you are authoring a library, then you might find yourself in the need for creating one.

The Promise global also provides a bunch of utility methods, but we will talk about those later.

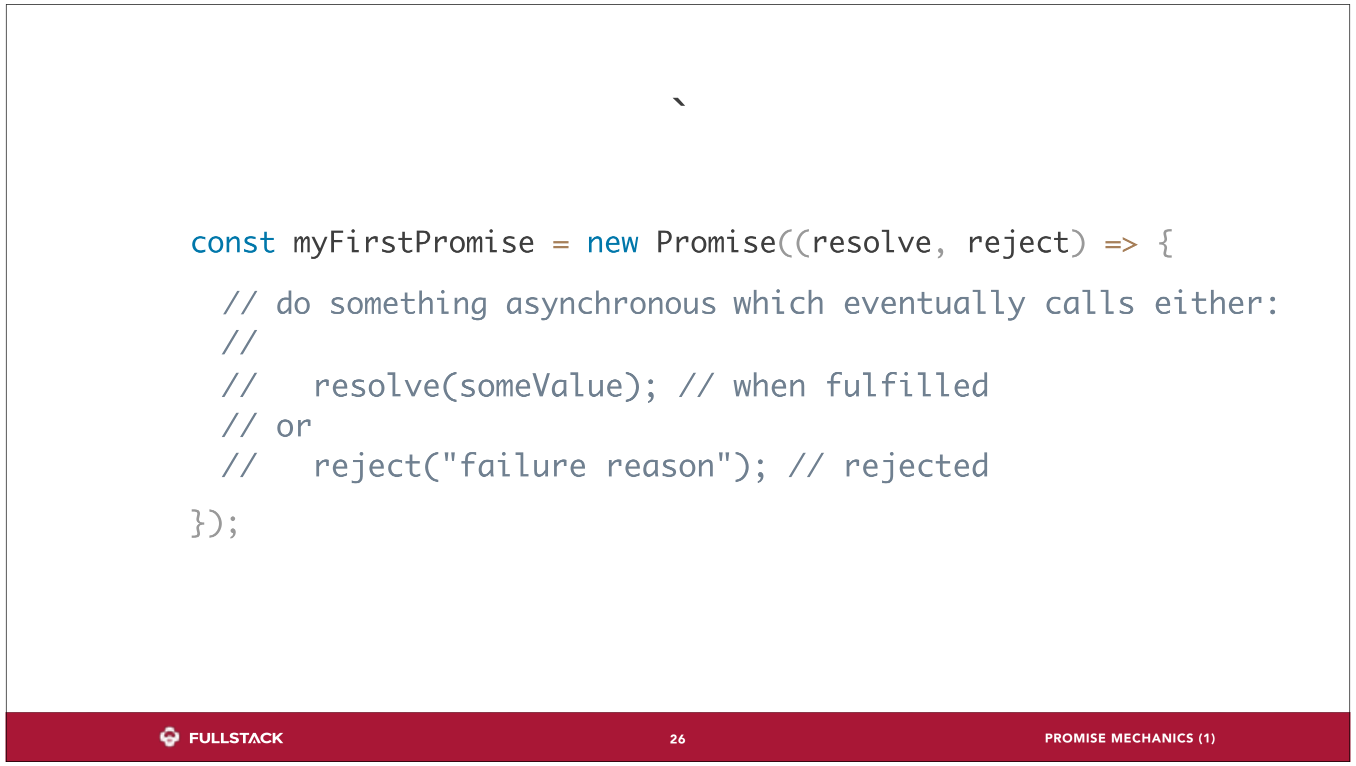


new Promise(executor)

```
const myFirstPromise = new Promise((resolve, reject) => {
  // do something asynchronous
});
```

Executor Function

This constructor takes as its argument a function, called the "executor function”.
This function receives `resolve` and `reject` functions which let you control the promise's eventual fate.



`

```
const myFirstPromise = new Promise((resolve, reject) => {
  // do something asynchronous which eventually calls either:
  //
  //   resolve(someValue); // when fulfilled
  // or
  //   reject("failure reason"); // rejected
});
```

Show example of promisifying readFile ourselves.
This is NOT something that you need to memorize, but you should know how to look it up in case you ever need to to it. This is essentially what the ‘promisify’ function in Node.js is doing.

Lab