Mechanics of Promises *(1)*

Understanding JavaScript Promise Generation & Behavior

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

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

(PROMISES)

```
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")

}
```

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

(PROMISES)

```
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")
}
```

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

(PROMISES)

```
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")

}
```

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

(PROMISES)

```
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")
}
```

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

(PROMISES)

```
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")

}
```

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

(PROMISES)

```
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")
}
```

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

(PROMISES)

```
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")

}
```

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

(PROMISES)

```
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")

}
```

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

(PROMISES)

```
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")

}
```

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

(PROMISES)

```
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")

}
```

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

(PROMISES)

```
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")

}
```

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

(PROMISES)

```
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")
}
```

PROMISE ADVANTAGES

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

Promises can be passed around...

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

Promises can be passed around…

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

Promises are portable…

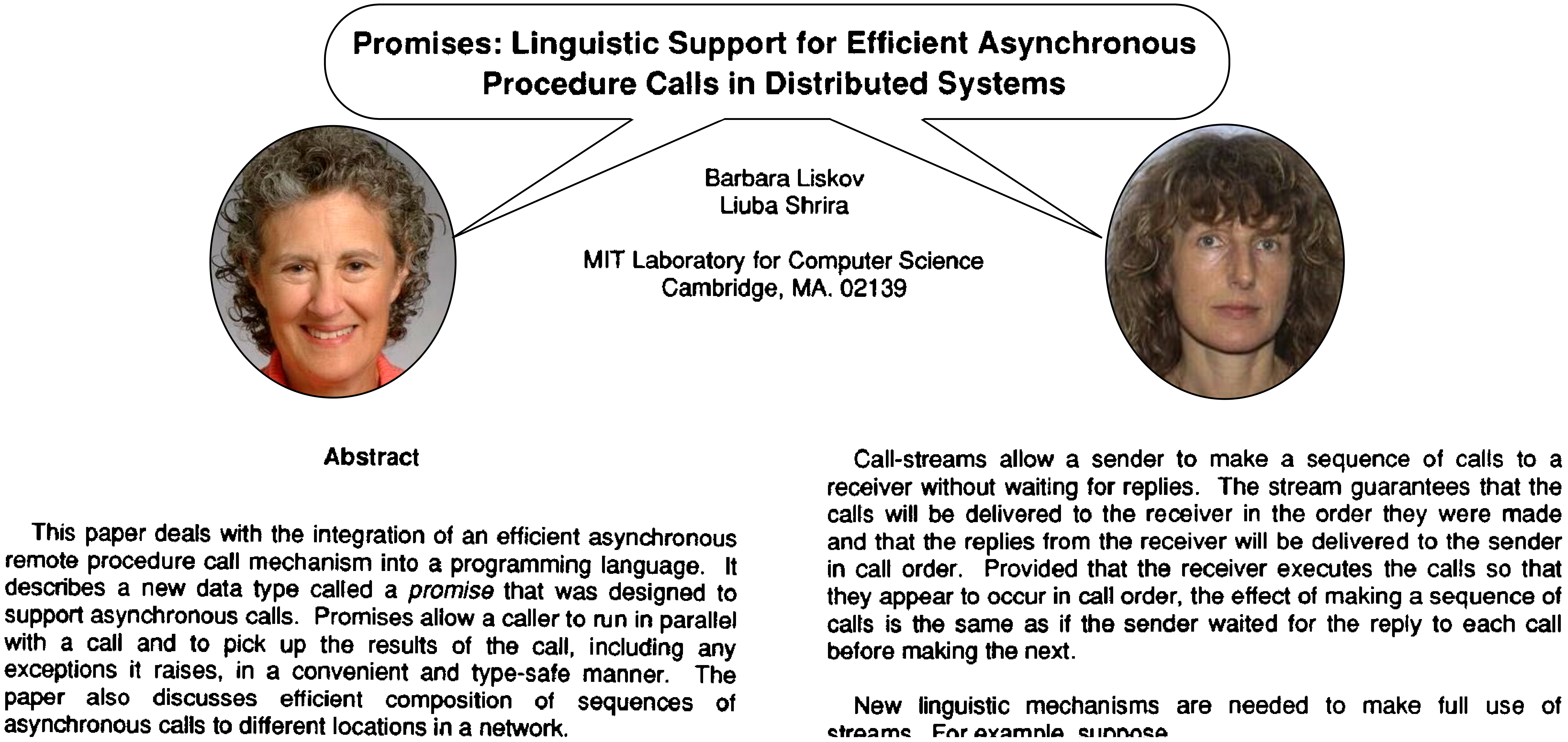
```
let lucky = readFileAsync('/luckyNumber.txt')

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

Export to other modules...

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

How Did We End Up Here?



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

Barbara Liskov
Liuba Shrira

MIT Laboratory for Computer Science
Cambridge, MA. 02139

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

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

PROMISES IN JS: TIMELINE

2009
Node
2011
jQuery 1.5
2014/15
ES6
2010
Q.js
A+
2013
Bluebird
2011
When.js
2008
DOJO

So, what is a promise?

promises are objects

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.

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 Outcomes

Promise Outcomes

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

Promise Outcomes

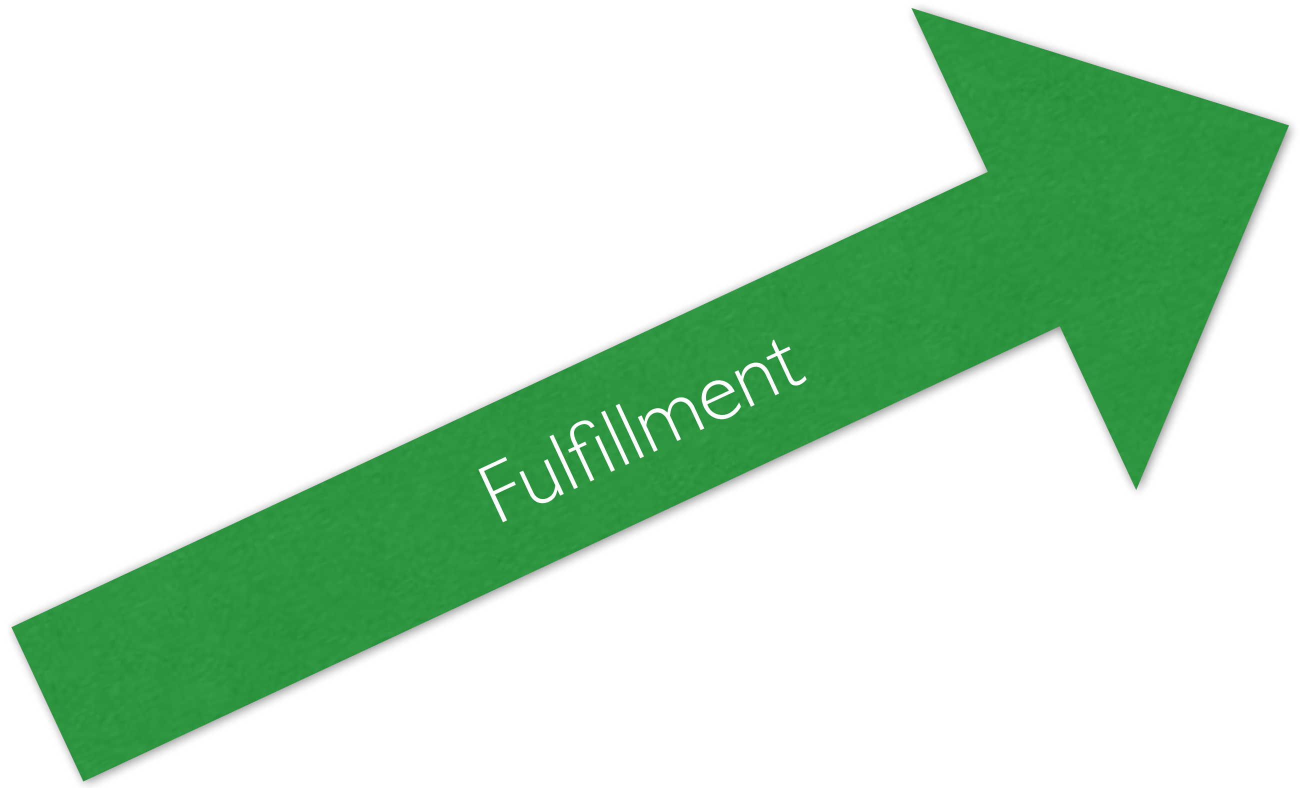
```
readFileAsync('/luckyNumber.txt')
```

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

Promise Outcomes

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

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



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

Promise Outcomes

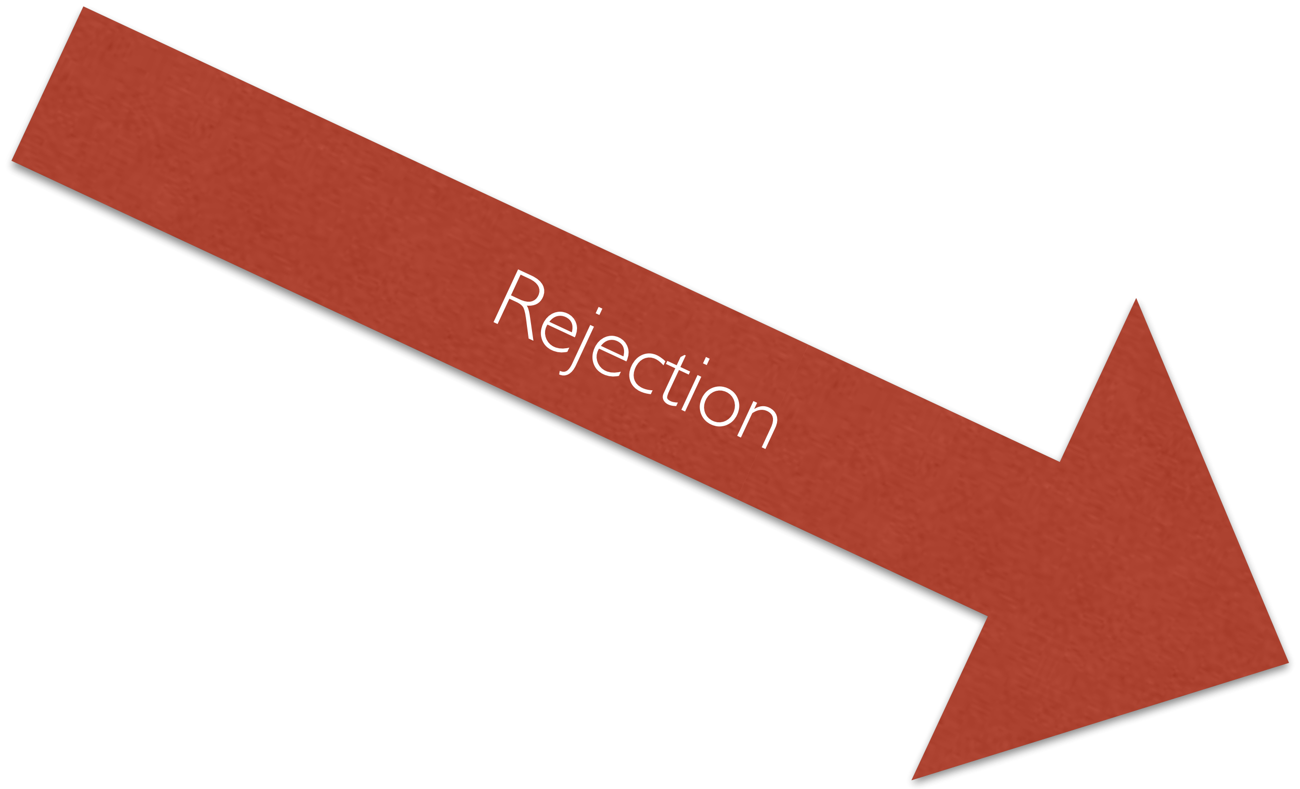
```
readFileAsync('/luckyNumber.txt')
```

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

Promise Outcomes

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

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



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

Promise Outcomes

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.

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);
```

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..

new Promise(executor)

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

new Promise(executor)

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

Executor Function

new Promise(executor)

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

new Promise

```
const myFirstPromise = new Promise((resolve, reject) => {

  // do something asynchronous which eventually calls either:
  //
  //   resolve(someValue); // when fulfilled
  // or
  //   reject(Error("failure reason")); // rejected

});
```

Lab