Promises

Lecture 9: Building Modern Web Applications

Karthik Pattabiraman
Kumseok Jung

What is a Promise

1. **What is a Promise**

2. How to use Promises

3. Asynchronous Programming with Promises

What is a Promise

- Promise is a new built-in object **introduced in ES6**
- Provides a **cleaner interface** for handling **asynchronous operations**
- When multiple asynchronous operations need to be made, the **callback** pattern **becomes hard to follow**
  - Scope of variables in multiple nested closures
  - Error handling for each of the callback steps

Why use Promise?

- Consider a function `first` with the following signature:
  - `function first(arg, callback)`
  - `arg` is some data
  - `callback` is a function accepting 2 arguments: `error` and `result`

↳ calls it after doing some async stuff.

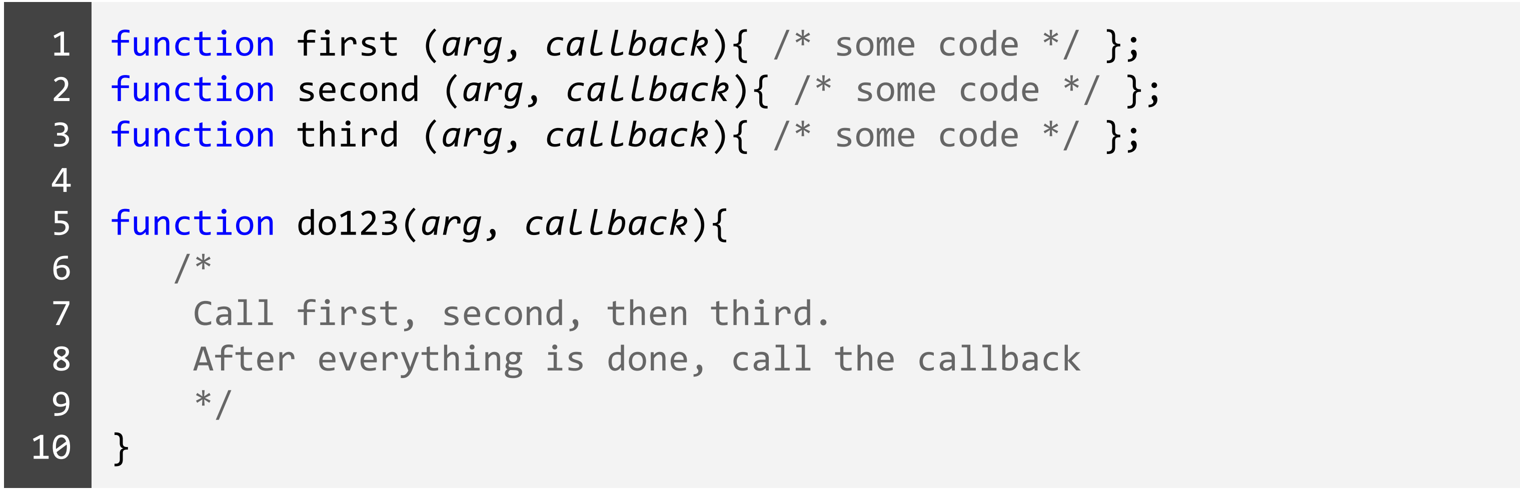
```
function first (arg, callback){
   var result = null;
   // do some asynchronous stuff ...
      callback(result);
   // ... do some other stuff
}

first("Hello World", (error, result)=> {
   console.log(error ? "ERROR!" : result);
});
```

arrow fx

Common pattern in node.Js.
use as callback

Why use Promise?

- Consider 2 more functions with similar function signatures:
  - `function second(arg, callback)`
  - `function third(arg, callback)`
- How to create a new function that calls the 3 functions in sequence?

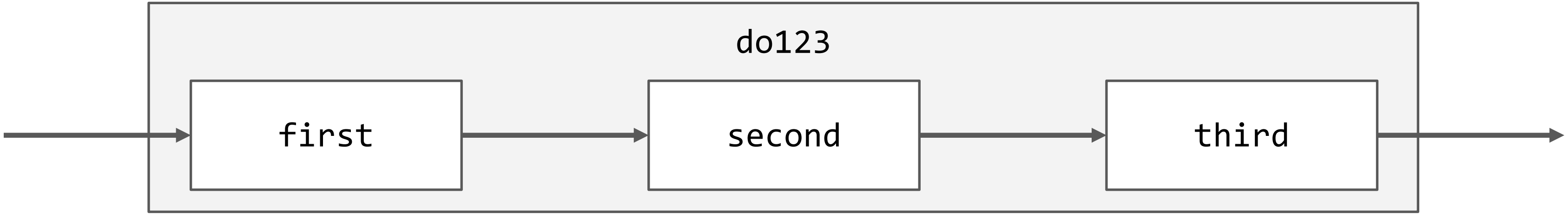


```
function first (arg, callback){ /* some code */ };
function second (arg, callback){ /* some code */ };
function third (arg, callback){ /* some code */ };

function do123(arg, callback){
   /*
    Call first, second, then third.
    After everything is done, call the callback
    */
}
```

Why use Promise?

- Consider 2 more functions with similar function signatures:
  - `function second(arg, callback)`
  - `function third(arg, callback)`
- How to create a new function that calls the 3 functions in sequence?

Sequentially



do123

first

second

third

- Asynchronously
- Can't just call first, second, third; event handler calls randomly

Why use Promise?

- How to create a new function that calls the 3 functions in sequence?

```
function do123(arg, callback){







}



```

Why use Promise?

- How to create a new function that calls the 3 functions in sequence?

```
function do123(arg, callback){
   first(arg, (err1, result1)=> {






   });
}



```

Why use Promise?

- How to create a new function that calls the 3 functions in sequence?

```
function do123(arg, callback){
   first(arg, (err1, result1)=> {
      second(result1, (err2, result2)=> {



      });
   });
}
```

Why use Promise?

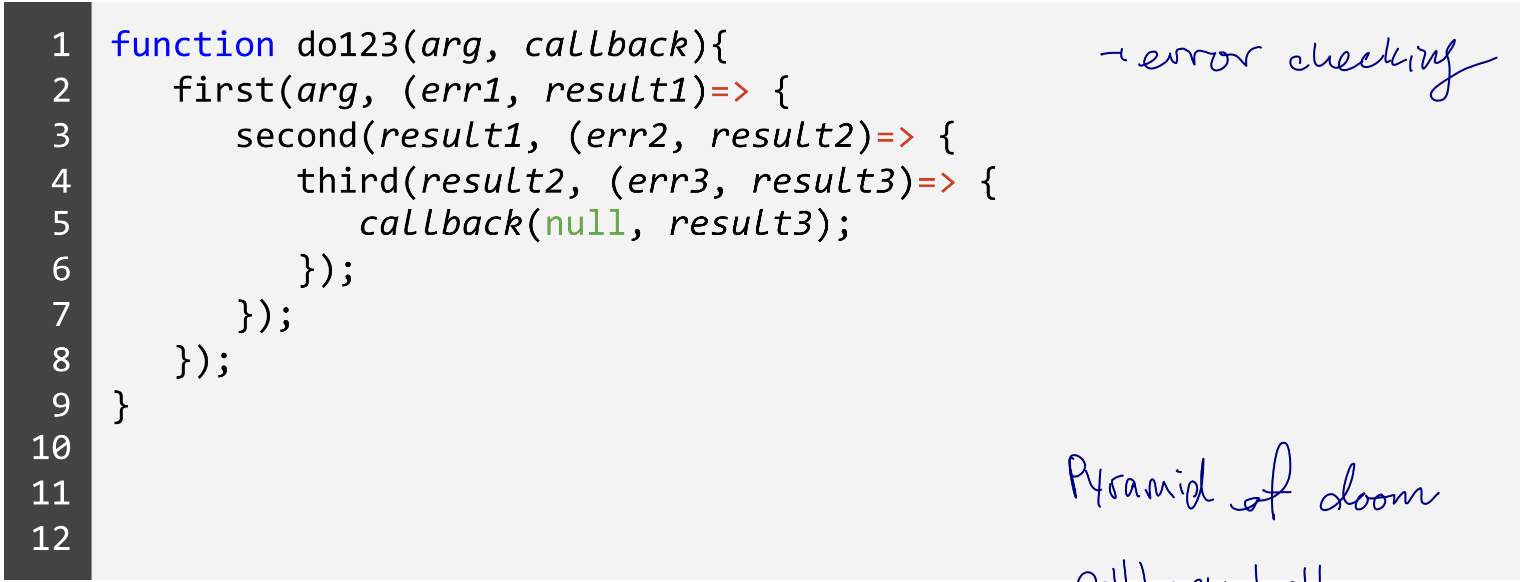
- How to create a new function that calls the 3 functions in sequence?

```
function do123(arg, callback){
   first(arg, (err1, result1)=> {
      second(result1, (err2, result2)=> {
         third(result2, (err3, result3)=> {

         });
      });
   });
}
```

- fairly typical.
- call in callback
- hard to understand
- each can return error.

Why use Promise?

- How to create a new function that calls the 3 functions in sequence?



```
function do123(arg, callback){
   first(arg, (err1, result1)=> {
      second(result1, (err2, result2)=> {
         third(result2, (err3, result3)=> {
            callback(null, result3);
         });
      });
   });
}
```

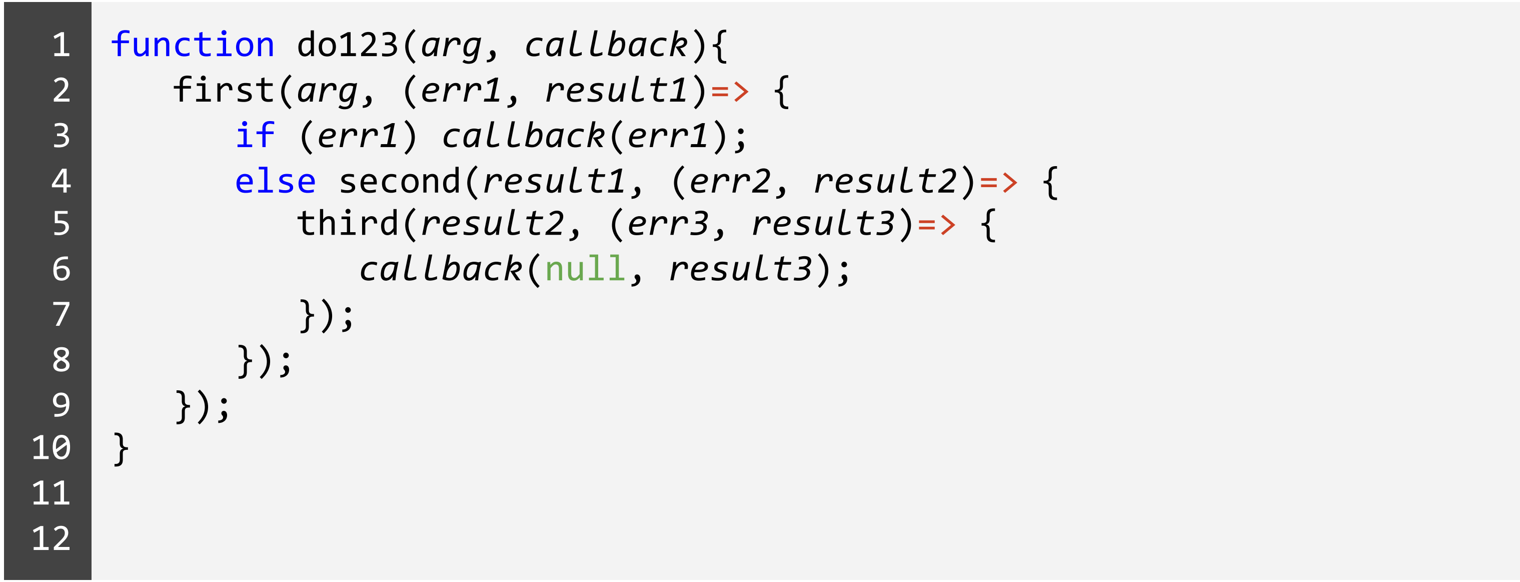
+ error checking

Pyramid of doom

callback hell

Why use Promise?

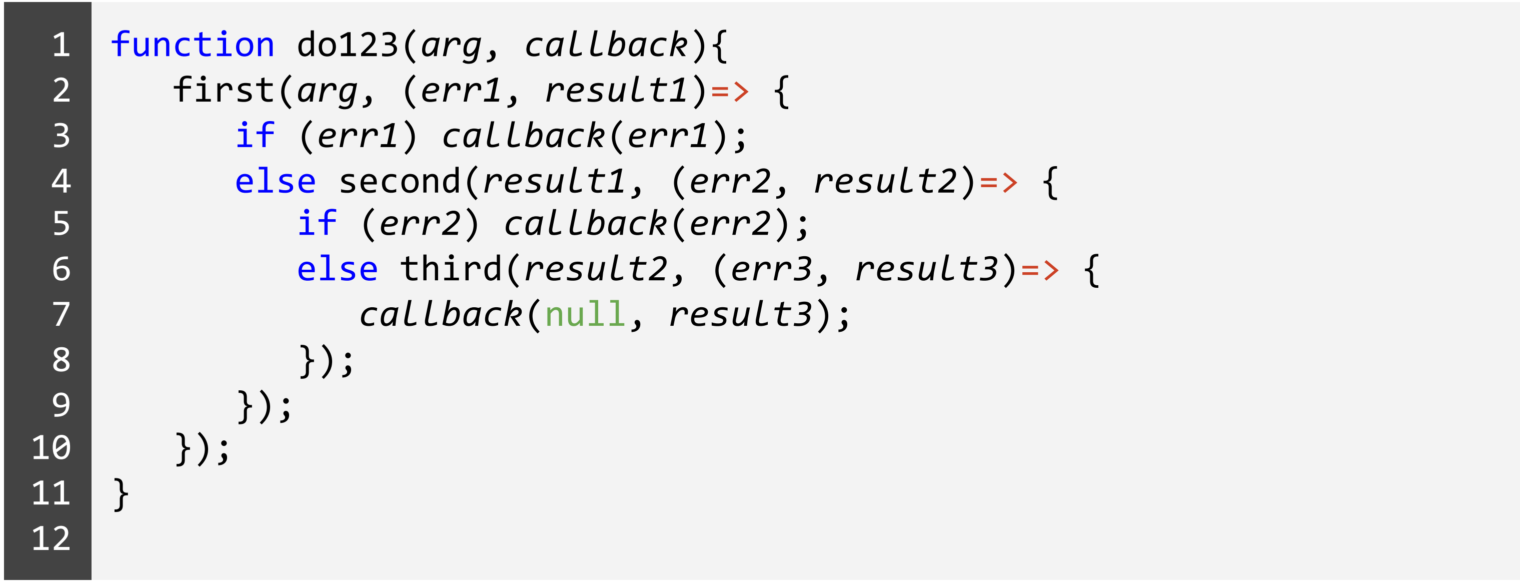
- How to create a new function that calls the 3 functions in sequence?



```javascript
function do123(arg, callback){
   first(arg, (err1, result1)=> {
      if (err1) callback(err1);
      else second(result1, (err2, result2)=> {
         third(result2, (err3, result3)=> {
            callback(null, result3);
         });
      });
   });
}
```

Why use Promise?

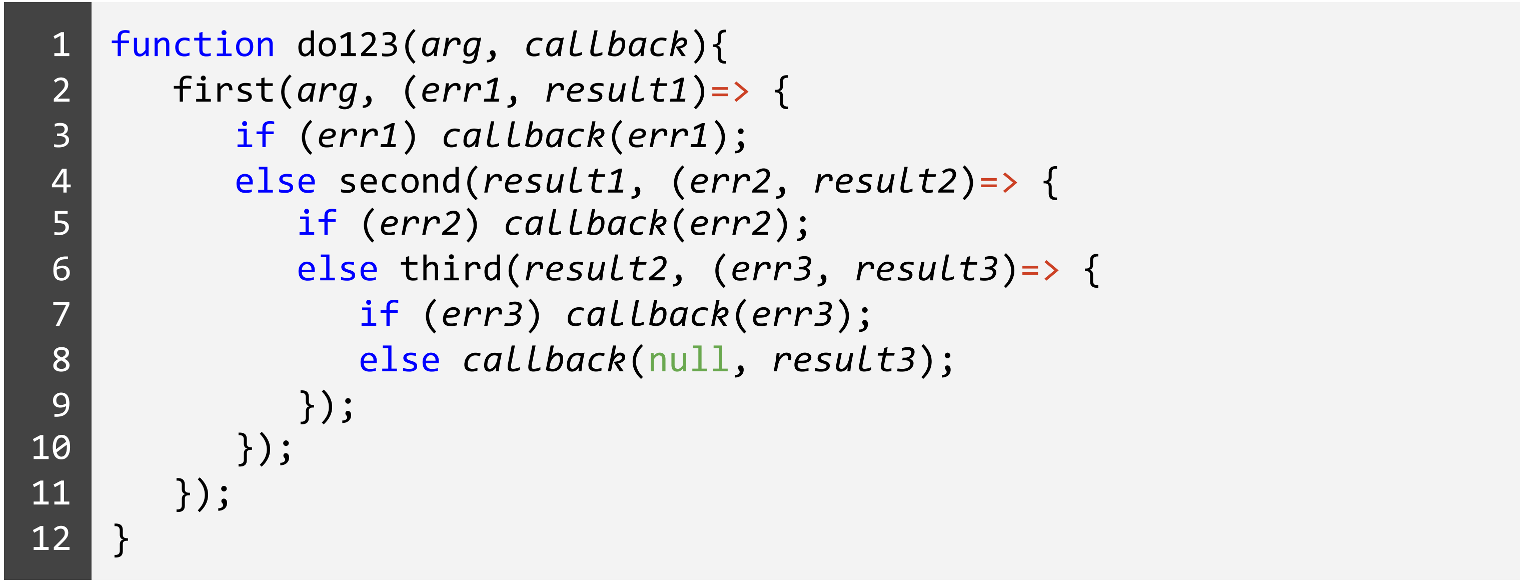
- How to create a new function that calls the 3 functions in sequence?



```
function do123(arg, callback){
   first(arg, (err1, result1)=> {
      if (err1) callback(err1);
      else second(result1, (err2, result2)=> {
         if (err2) callback(err2);
         else third(result2, (err3, result3)=> {
            callback(null, result3);
         });
      });
   });
}
```

Why use Promise?

- How to create a new function that calls the 3 functions in sequence?



```
function do123(arg, callback){
   first(arg, (err1, result1)=> {
      if (err1) callback(err1);
      else second(result1, (err2, result2)=> {
         if (err2) callback(err2);
         else third(result2, (err3, result3)=> {
            if (err3) callback(err3);
            else callback(null, result3);
         });
      });
   });
}
```

Why use Promise?

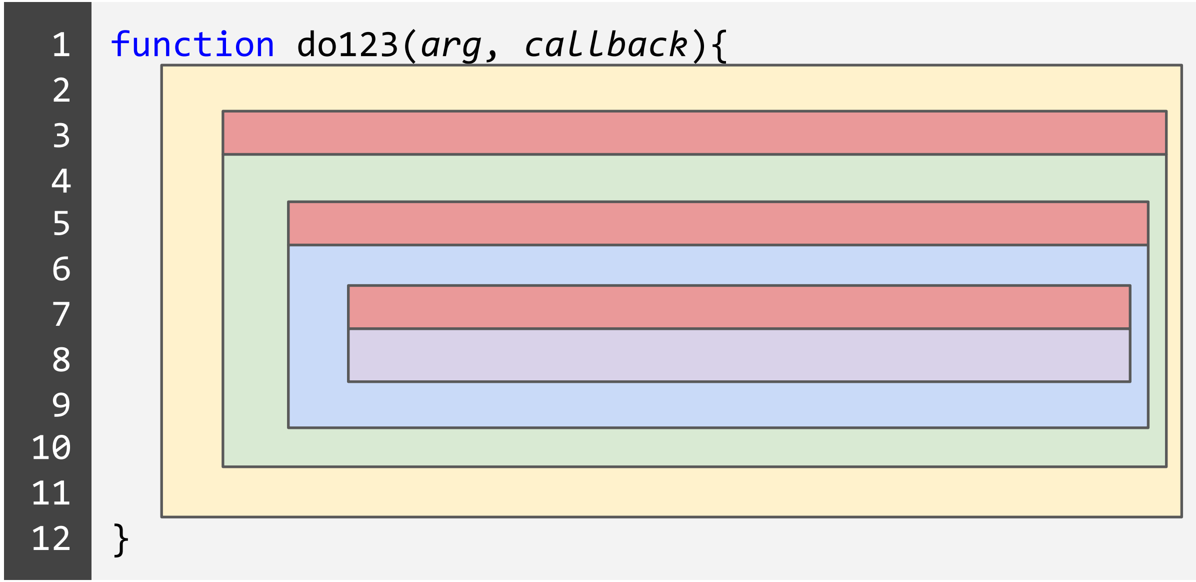
- How to create a new function that calls the 3 functions in sequence?

```
function do123(arg, callback){
   first(arg, (err1, result1)=> {
      if (err1) callback(err1);
      else second(result1, (err2, result2)=> {
         if (err2) callback(err2);
         else third(result2, (err3, result3)=> {
            if (err3) callback(err3);
            else callback(null, result3);
         });
      });
   });
}
```

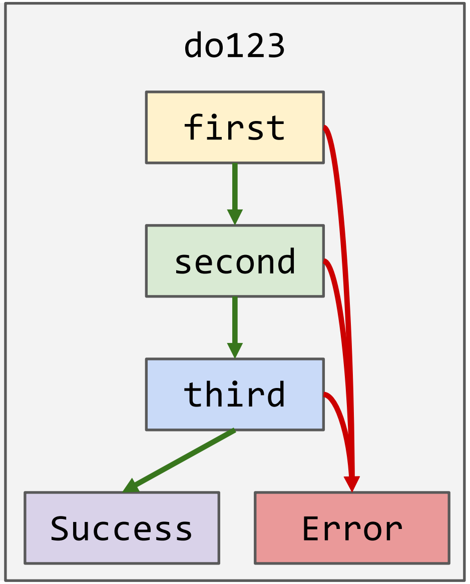
Callback Hell

Why use Promise?

- Problem with callbacks: the **code structure does not follow** the **logical structure**

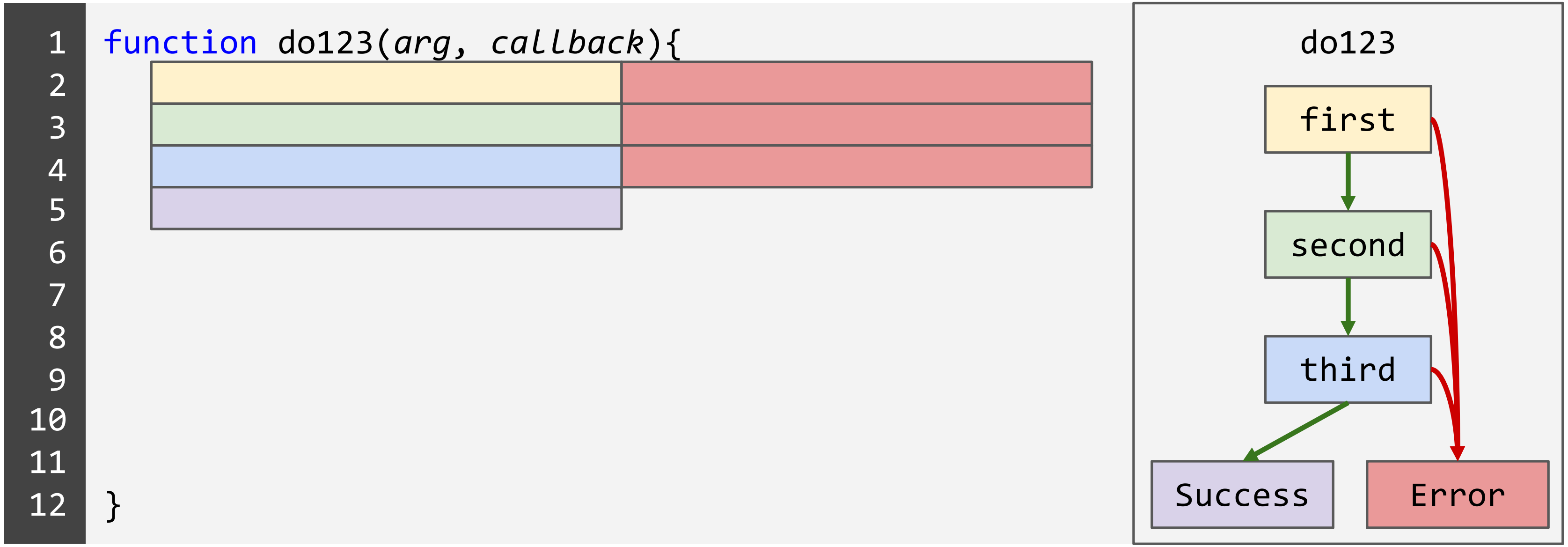


```
function do123(arg, callback){











}
```



Why use Promise?

- It would be nice if the **code structure followed** the **logical structure**



```
function do123(arg, callback){










}
```

Why use Promise?

- Consider the same `first` function using a `Promise`-based interface
  - `function first(arg)` - notice the lack of a `callback` argument
  - `arg` is some data
  - returns a `Promise` object

Promise done

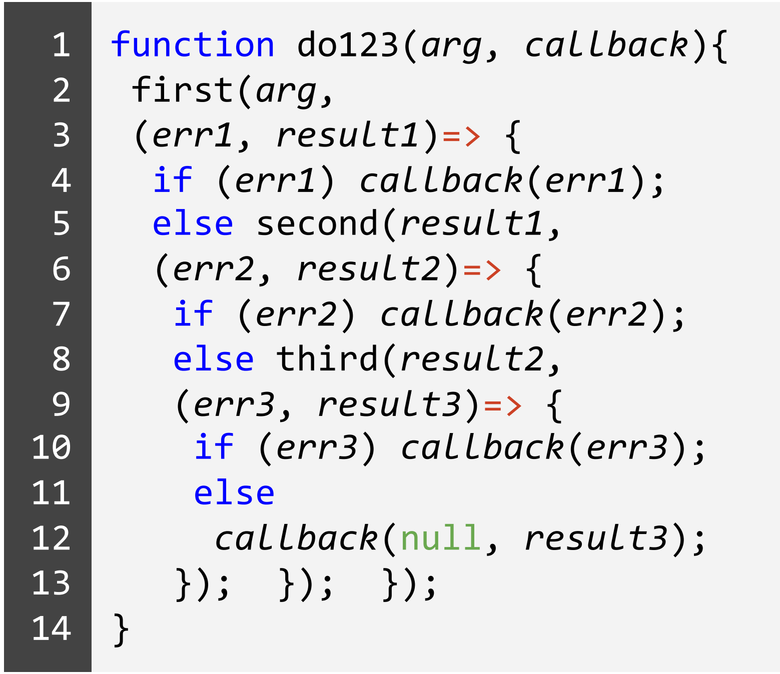
did not complete.

```
function first (arg){
   return new Promise((resolve, reject)=> {
      var result = null;
      // do some asynchronous stuff ...
         resolve(result);
      // ... do some other stuff
   });
}
first("Hello World")
 .then(console.log, (error)=> console.log("ERROR!"));
```

Why use Promise?

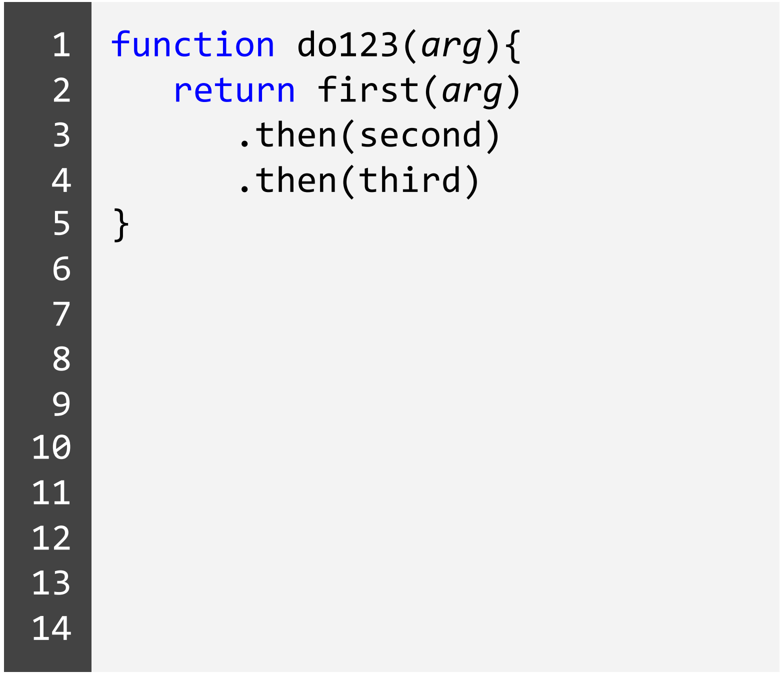
elegant

Using ES5 Callbacks



```
function do123(arg, callback){
 first(arg,
 (err1, result1)=> {
  if (err1) callback(err1);
  else second(result1,
  (err2, result2)=> {
   if (err2) callback(err2);
   else third(result2,
   (err3, result3)=> {
    if (err3) callback(err3);
    else
     callback(null, result3);
   });  });  });
}
```

Using ES6 Promises



```
function do123(arg){
   return first(arg)
      .then(second)
      .then(third)
}
```

How to use Promises

1. What is a Promise

2. **How to use Promises**

3. Asynchronous Programming with Promises

Promise

- `Promise` is an object with the following methods
  - `then (onResolve, onReject)`: used to register resolve and reject callbacks
  - `catch (onReject)`: used to register reject callback
  - `finally (onComplete)`: used to register settlement callback
- `Promise` will be in one of the three states: pending, resolved, rejected

error, not completed.

Action completed

- `Promise` also has static methods
  - `resolve (value)`: returns a `Promise` that resolves immediately to `value`
  - `reject (error)`: returns a `Promise` that rejects immediately to `error`
  - `all (promises)`: returns a `Promise` that resolves when all promises resolve
  - `race (promises)`: returns a `Promise` that resolves if any of the promises resolve

- Cannot revert state, only one way
- typically, not always pending → resolve.

Promise

- Creating a `Promise` object
  - `new Promise(func)`: The `Promise` constructor expects a single argument *func*, which is a function with 2 arguments: `resolve`, `reject`
  - `resolve` and `reject` are callback functions for emitting the result of the operation
    - `resolve(result)` to emit the result of a successful operation
    - `reject(error)` to emit the error from a failed operation

callback → of promise object

```
var action = new Promise((resolve, reject)=> {
   var result = null;
   // do some asynchronous stuff ...
   if (noError) resolve(result);
   else reject(new Error("Something Wrong"));
   // ... do some other stuff
});
```

- How to resolve a promise
reject.

if you call resolve, then goes
to resolve state

Promise

- Creating a `Promise` object
  - `new Promise(func)`: The `Promise` constructor expects a single argument *`func`*, which is a function with 2 arguments: `resolve`, `reject`
  - `resolve` and `reject` are callback functions for emitting the result of the operation
    - `resolve(result)` to emit the result of a successful operation
    - `reject(error)` to emit the error from a failed operation

-Promise gives you two fx to deal.

```
var action = new Promise((resolve, reject)=> {
   setTimeout(()=> {
      if (Math.random() > 0.5) resolve("Success!");
      else reject(new Error("LowValueError"));
   }, 1000);
});

```

← Async Timout.

Promise

- Using the result of a `Promise` fulfillment through the `then` method
  - `then(onResolve, onReject)`: used to register callbacks for handling the result of the `Promise`. It returns another `Promise`, making this function **chainable**
  - `onResolve` is called **if the previous `Promise` resolves**; it receives the resolved value as the only argument
  - `onReject` is called **if the previous `Promise` rejects** or **throws an error**; it receives the rejected value or the error object as the only argument

```
action.then(
    (result)=> console.log(result), // result: "Success!"
    (error)=> console.log(error)    // error: Error("LowValueError")
);
```

first arg

resolve

→ returns Promise obj.

↳ useful for chain.

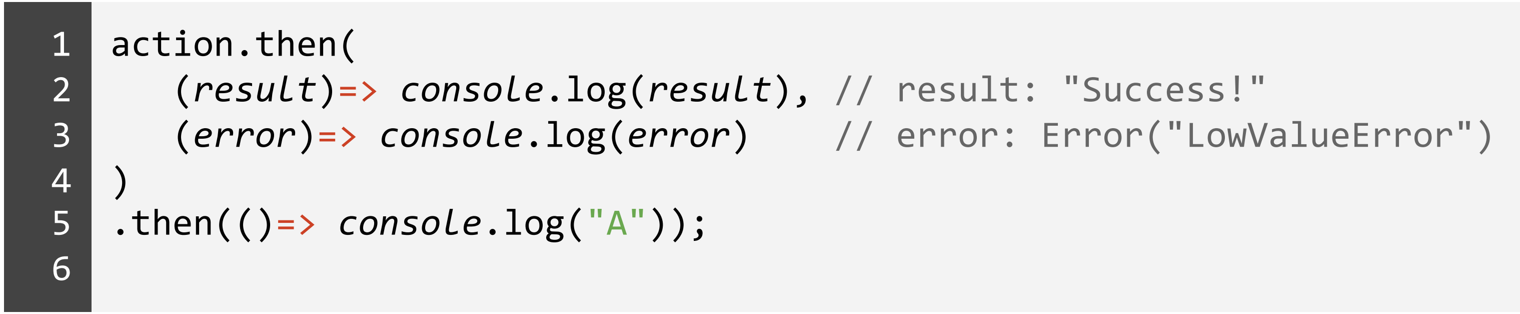
Depends on resolve or rej

second arg

→ reject.

Promise

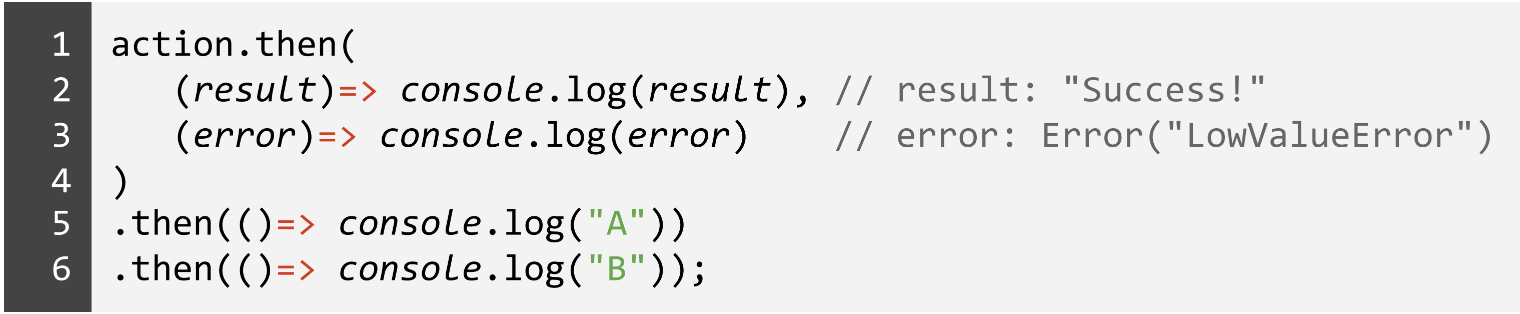
- Using the result of a `Promise` fulfillment through the `then` method
  - `then(onResolve, onReject)`: used to register callbacks for handling the result of the `Promise`. It returns another `Promise`, making this function **chainable**
  - `onResolve` is called **if the previous `Promise` resolves**; it receives the resolved value as the only argument
  - `onReject` is called **if the previous `Promise` rejects** or **throws an error**; it receives the rejected value or the error object as the only argument



```
action.then(
    (result)=> console.log(result), // result: "Success!"
    (error)=> console.log(error)    // error: Error("LowValueError")
)
.then(()=> console.log("A"));

```

Promise

- Using the result of a `Promise` fulfillment through the `then` method
  - `then(onResolve, onReject)`: used to register callbacks for handling the result of the `Promise`. It returns another `Promise`, making this function **chainable**
  - `onResolve` is called **if the previous `Promise` resolves**; it receives the resolved value as the only argument
  - `onReject` is called **if the previous `Promise` rejects** or **throws an error**; it receives the rejected value or the error object as the only argument



```
action.then(
   (result)=> console.log(result), // result: "Success!"
   (error)=> console.log(error)    // error: Error("LowValueError")
)
.then(()=> console.log("A"))
.then(()=> console.log("B"));
```

Class Activity: Promise Chaining

lectures/lecture-9/activity1.js

- Create a `resolveAfter` function that resolves after a specified amount of *`time`*, returning a `Promise` object
  - The function should print the given `time` before resolving
- Using the `resolveAfter` function and the `then` method to chain the promises, make the program print 500, 1000, 1500 one after another

```
function resolveAfter (time){
   // to implement
}

resolveAfter(500)
.then(/* to implement */)
```

Promise

- The `catch` method is used to handle the result of a rejected `Promise`
  - `catch(onReject)`: used to register a callback for handling the result of the failed `Promise`. It returns another `Promise`, making this function **chainable**
  - `onReject` is called **if the previous `Promise` rejects** or **throws an error**; it receives the rejected value or the error object as the only argument

```
action.then(
   (result)=> console.log(result), // result: "Success!"
   (error)=> console.log(error)    // error: Error("LowValueError")
)
.catch((err)=> console.log(err));
```

if ERR

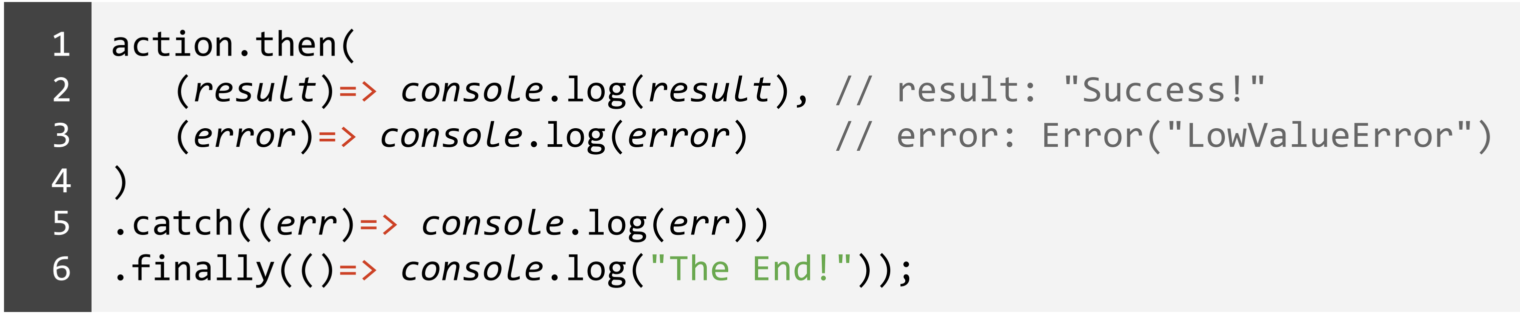
(after reject code runs)

Add at end of chain - If any reject, send to catch

Promise

- The finally method is used to register a callback to be called when a Promise is settled, regardless of the result → success or fail.
    - finally(onComplete): It returns another Promise, making this function **chainable**
    - onComplete is called **if the previous Promise is settled**

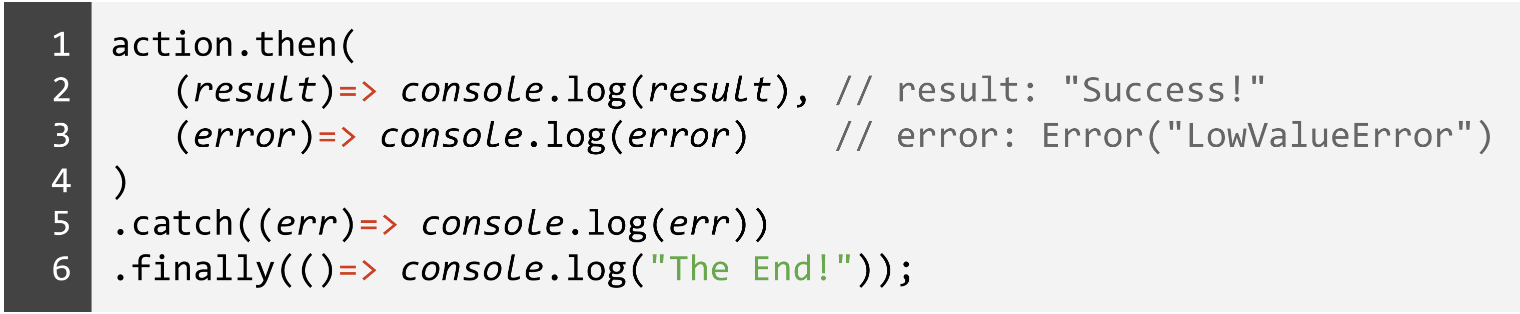
- Only one finally at end, for cleanup only



```
action.then(
    (result)=> console.log(result), // result: "Success!"
    (error)=> console.log(error)    // error: Error("LowValueError")
)
.catch((err)=> console.log(err))
.finally(()=> console.log("The End!"));
```

Promise

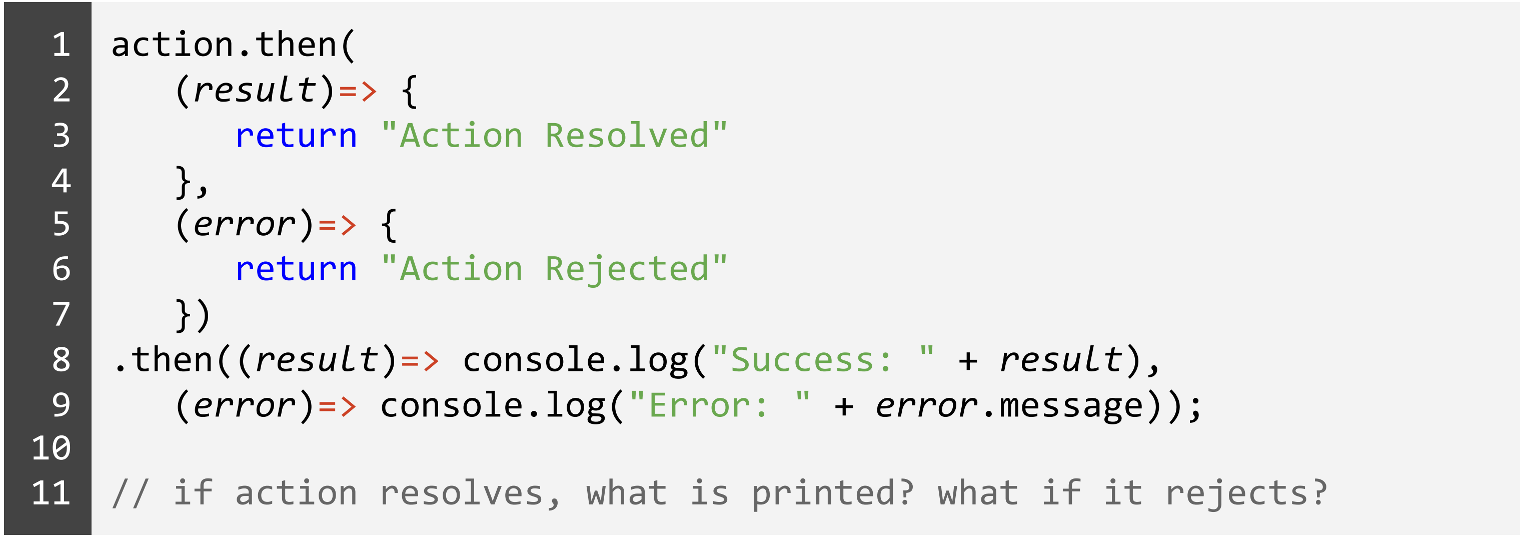
- The static functions `Promise.resolve` and `Promise.reject` are used to create a `Promise` object that immediately resolves or rejects with the given data
    - Useful when the next asynchronous operation expects a `Promise` object



```
action.then(
    (result)=> console.log(result), // result: "Success!"
    (error)=> console.log(error)    // error: Error("LowValueError")
)
.catch((err)=> console.log(err))
.finally(()=> console.log("The End!"));
```

Promise

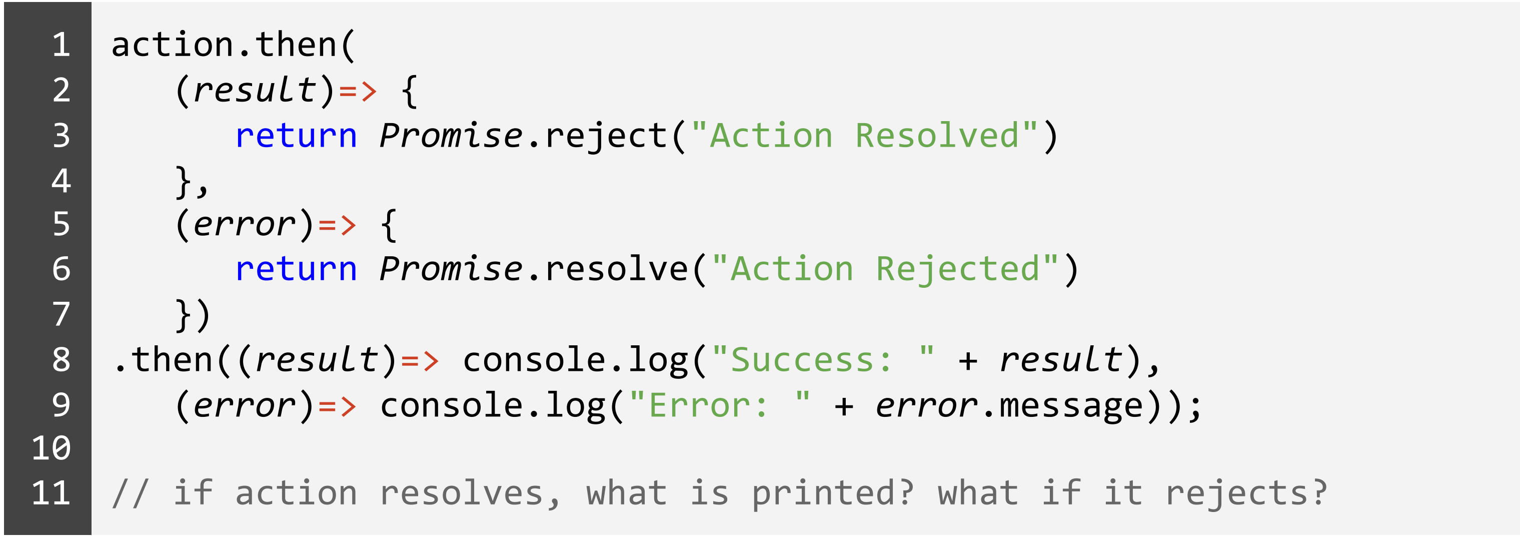
- The return values of the callback functions given to `then`, `catch`, and `finally` method are wrapped as a resolved `Promise`, if it is not already a `Promise`



```javascript
action.then(
   (result)=> {
      return "Action Resolved"
   },
   (error)=> {
      return "Action Rejected"
   })
.then((result)=> console.log("Success: " + result),
   (error)=> console.log("Error: " + error.message));

// if action resolves, what is printed? what if it rejects?
```

Promise

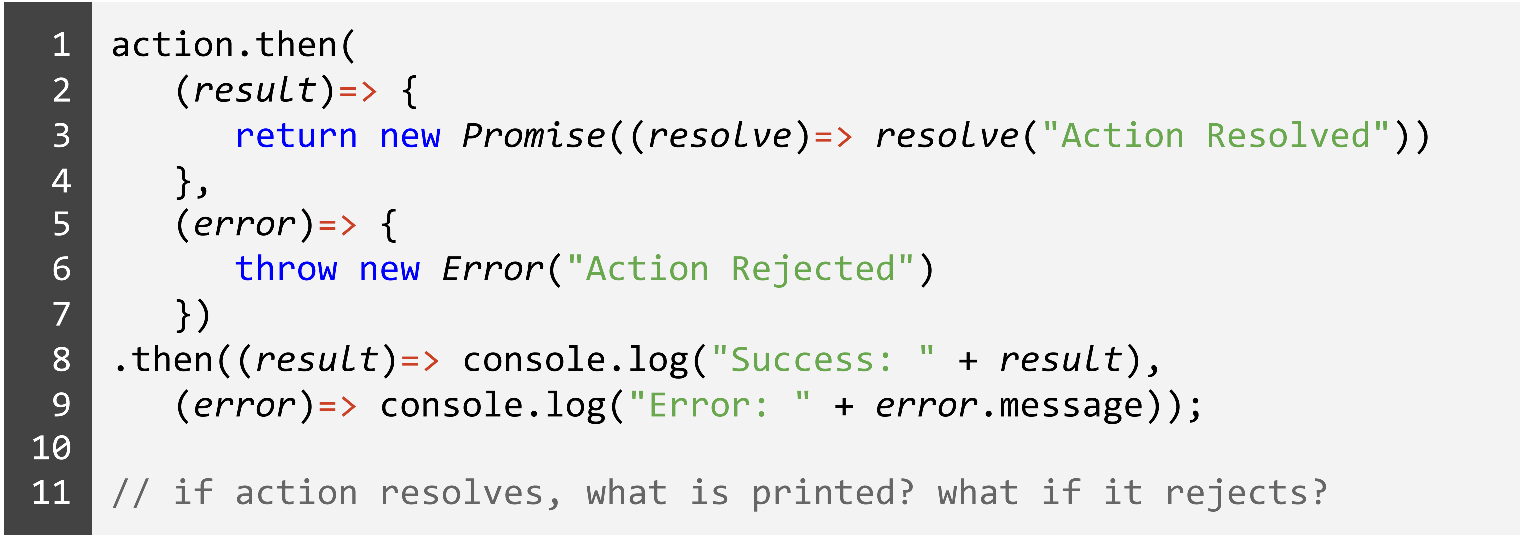
- The return values of the callback functions given to `then`, `catch`, and `finally` method are wrapped as a resolved `Promise`, if it is not already a `Promise`



```
action.then(
   (result)=> {
      return Promise.reject("Action Resolved")
   },
   (error)=> {
      return Promise.resolve("Action Rejected")
   })
.then((result)=> console.log("Success: " + result),
   (error)=> console.log("Error: " + error.message));

// if action resolves, what is printed? what if it rejects?
```

Promise

- The return values of the callback functions given to `then`, `catch`, and `finally` method are wrapped as a resolved `Promise`, if it is not already a `Promise`



```
action.then(
   (result)=> {
      return new Promise((resolve)=> resolve("Action Resolved"))
   },
   (error)=> {
      throw new Error("Action Rejected")
   })
.then((result)=> console.log("Success: " + result),
   (error)=> console.log("Error: " + error.message));

// if action resolves, what is printed? what if it rejects?
```

Class Activity: Promisify

lectures/lecture-9/activity2.js

- Create a `readFile` function that wraps the Node.js `fs.readFile` function and provides a `Promise`-based interface
  - `function readFile(filepath)`
  - returns a `Promise` object that resolves to the file content, or rejects if error occurred

```
var fs = require("fs");     // you can use fs.readFile

function readFile (filepath){
   // to implement
}

readFile("example.txt")
.then((result)=> console.log(result.length))
.catch((error)=> console.log(error));

```

How to use Promises

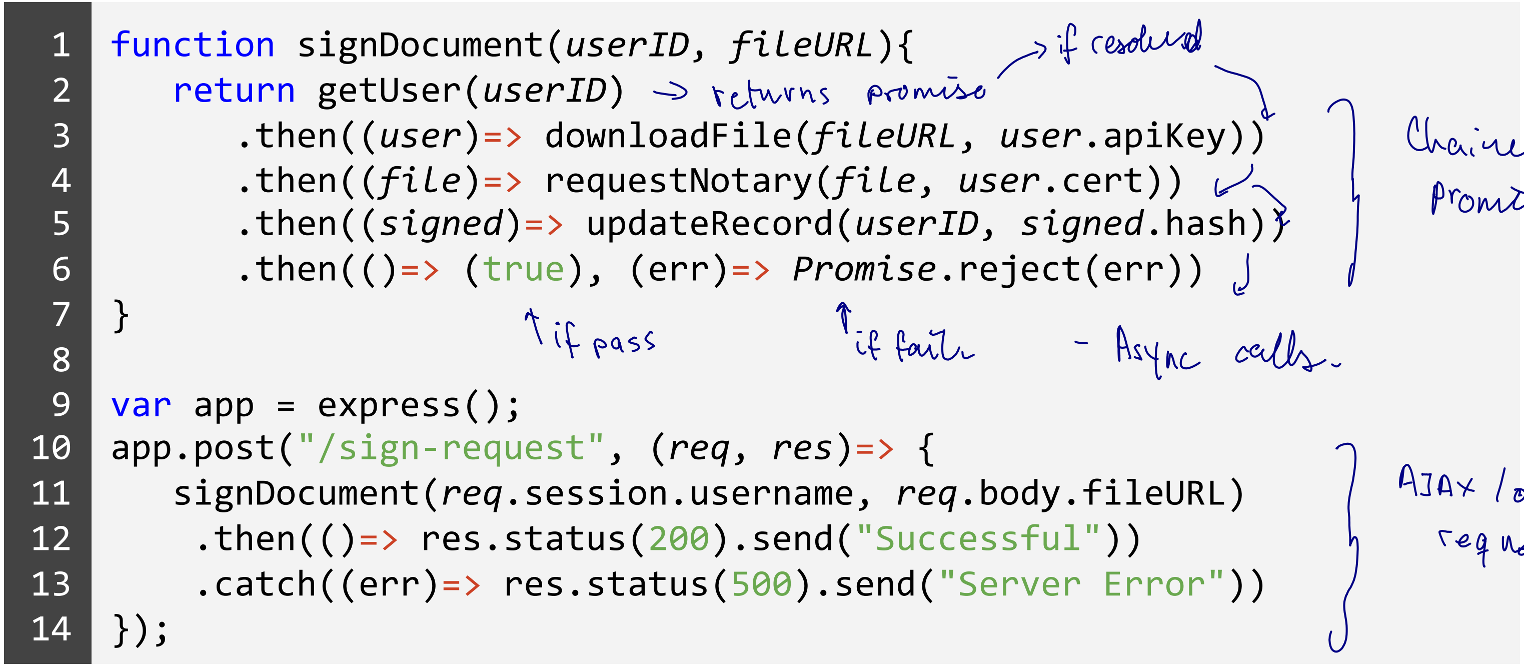
1. What is a Promise
2. How to use Promises
3. **Asynchronous Programming with Promises**

Asynchronous Programming

- JavaScript involves a lot of asynchronous operations
  - The Internet is where JavaScript is used: this involves a lot of **AJAX requests**
  - The **I/O model** for the JavaScript VM is **asynchronous**: files, sockets, processes, Inter-process communication, and I/O streams all handled by **asynchronous API**
- The `Promise` API makes it easy to compose a sequence of asynchronous operations as a dataflow pipeline

Asynchronous Programming

Example: Node.js application providing a document signing service



```
function signDocument(userID, fileURL){
   return getUser(userID)
      .then((user)=> downloadFile(fileURL, user.apiKey))
      .then((file)=> requestNotary(file, user.cert))
      .then((signed)=> updateRecord(userID, signed.hash))
      .then(()=> (true), (err)=> Promise.reject(err))
}

var app = express();
app.post("/sign-request", (req, res)=> {
   signDocument(req.session.username, req.body.fileURL)
     .then(()=> res.status(200).send("Successful"))
     .catch((err)=> res.status(500).send("Server Error"))
});
```

→ returns promise → if resolved

Chained promise

↑ if pass

↑ if fail

- Async calls.

AJAX / other request.

Promise

- Using the static function `Promise.all`, we can wait for multiple concurrent `Promise`s to be resolved (sort of like joining threads)
    - `Promise.all` accepts an Array of promises and returns a `Promise` that resolves to an array of results (in the same order as the promises given)

```
var multi = Promise.all([
    new Promise((resolve)=> setTimeout(()=> resolve("A"), 2000)),
    new Promise((resolve)=> setTimeout(()=> resolve("B"), 3000)),
    new Promise((resolve)=> setTimeout(()=> resolve("C"), 1000)),
]);

multi.then(
    (results)=> console.log(results),
    (error)=> console.log(error));

```

↓
fails if any promise is rejected.

Promise

- Using the static function Promise.race, we can retrieve the first Promise to resolve out of a set of concurrent Promises
    - Promise.race accepts an Array of promises and returns the first Promise that resolves

→ OR operation.

```
var multi = Promise.race([
   new Promise((resolve)=> setTimeout(()=> resolve("A"), 2000)),
   new Promise((resolve)=> setTimeout(()=> resolve("B"), 3000)),
   new Promise((resolve)=> setTimeout(()=> resolve("C"), 1000)),
]);

multi.then(
   (result)=> console.log(result),
   (error)=> console.log(error));
```

- multiple copies, diffn latencies.

- if any promise reject, it will reject.

Class Activity

- Write a node.js program to read from two different text files and concatenate their contents using Promises. After both reads are complete, you should write the contents of the two files to a third file. You can assume that the order of reads is not important. You should not block for file read, nor read the files sequentially.

- How will you modify the above program if you wanted to write to the third file without waiting for both files to complete reading, again using promises ? Make sure that you follow the same constraints.