

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`



```
1 function first (arg, callback){
2   var result = null;
3   // do some asynchronous stuff ...
4   callback(result);
5   // ... do some other stuff
6 }
7
8 first("Hello World", (error, result)=> {
9   console.log(error ? "ERROR!" : result);
10 });
```

Why use Promise?

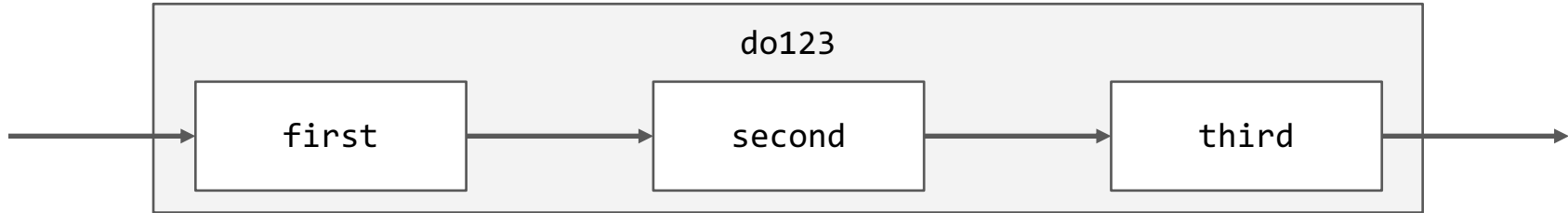
- Consider 2 more functions with similar function signatures:
 - `function second(arg, callback)`
 - `function third(arg, callback)`
- How to create a new function that calls the 3 functions in sequence?



```
1 function first (arg, callback){ /* some code */ };
2 function second (arg, callback){ /* some code */ };
3 function third (arg, callback){ /* some code */ };
4
5 function do123(arg, callback){
6     /*
7     Call first, second, then third.
8     After everything is done, call the callback
9     */
10 }
```

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?



Why use Promise?

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



```
1 function do123(arg, callback){  
2  
3  
4  
5  
6  
7  
8  
9 }  
10  
11  
12
```

Why use Promise?

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



```
1 function do123(arg, callback){  
2     first(arg, (err1, result1)=> {  
3  
4  
5  
6  
7  
8     });  
9 }  
10  
11  
12
```

Why use Promise?

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



```
1 function do123(arg, callback){  
2   first(arg, (err1, result1)=> {  
3     second(result1, (err2, result2)=> {  
4  
5  
6  
7     });  
8   });  
9 }  
10  
11  
12
```

Why use Promise?

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



```
1 function do123(arg, callback){  
2   first(arg, (err1, result1)=> {  
3     second(result1, (err2, result2)=> {  
4       third(result2, (err3, result3)=> {  
5  
6         });  
7       });  
8     });  
9   }  
10  
11  
12
```

Why use Promise?

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



```
1 function do123(arg, callback){
2   first(arg, (err1, result1)=> {
3     second(result1, (err2, result2)=> {
4       third(result2, (err3, result3)=> {
5         callback(null, result3);
6       });
7     });
8   });
9 }
10
11
12
```

Why use Promise?

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



```
1 function do123(arg, callback){
2   first(arg, (err1, result1)=> {
3     if (err1) callback(err1);
4     else second(result1, (err2, result2)=> {
5       third(result2, (err3, result3)=> {
6         callback(null, result3);
7       });
8     });
9   });
10 }
11
12
```

Why use Promise?

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



```
1 function do123(arg, callback){
2   first(arg, (err1, result1)=> {
3     if (err1) callback(err1);
4     else second(result1, (err2, result2)=> {
5       if (err2) callback(err2);
6       else third(result2, (err3, result3)=> {
7         callback(null, result3);
8       });
9     });
10  });
11 }
12
```

Why use Promise?

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



```
1 function do123(arg, callback){
2   first(arg, (err1, result1)=> {
3     if (err1) callback(err1);
4     else second(result1, (err2, result2)=> {
5       if (err2) callback(err2);
6       else third(result2, (err3, result3)=> {
7         if (err3) callback(err3);
8         else callback(null, result3);
9       });
10    });
11  });
12 }
```

Why use Promise?

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



```
1 function do123(arg, callback){
2   first(arg, (err1, result1)=> {
3     if (err1) callback(err1);
4     else second(result1, (err2, result2)=> {
5       if (err2) callback(err2);
6       else third(result2, (err3, result3)=> {
7         if (err3) callback(err3);
8         else callback(null, result3);
9       });
10    });
11  });
12 }
```

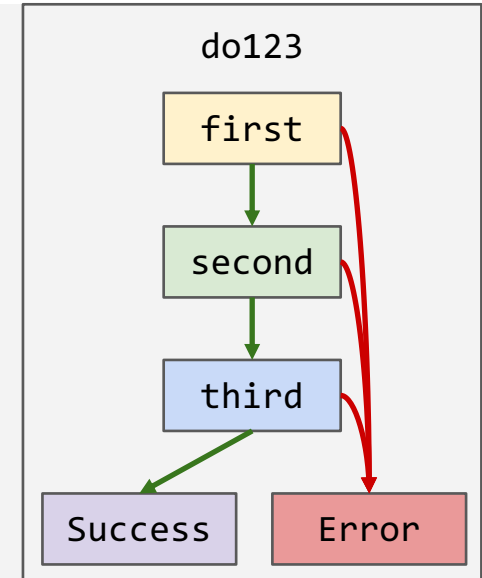
Callback Hell

Why use Promise?

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



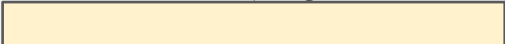
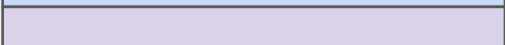
```
1 function do123(arg, callback){  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12 }
```

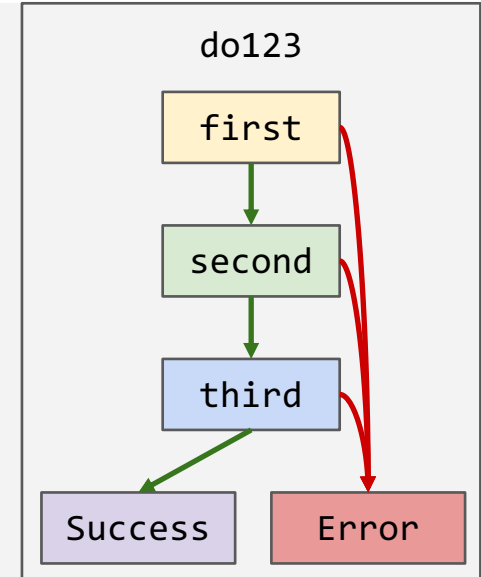
A diagram illustrating the code structure of a function with nested calls. It consists of four nested rectangles: a large yellow outer box, followed by a green box, a blue box, and a small purple box at the center. Each box contains a red horizontal bar, representing a function call or a step in a process. The nesting shows that each step is contained within the scope of the previous one, which is a common pattern in callback-based code.

Why use Promise?

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



```
1 function do123(arg, callback){  
2       
3       
4       
5       
6  
7  
8  
9  
10  
11  
12 }
```



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



```
1 function first (arg){
2   return new Promise((resolve, reject)=> {
3     var result = null;
4     // do some asynchronous stuff ...
5     resolve(result);
6     // ... do some other stuff
7   });
8 }
9 first("Hello World")
10 .then(console.log, (error)=> console.log("ERROR!"));
```

Why use Promise?

Using ES5 Callbacks

```
1 function do123(arg, callback){
2   first(arg,
3     (err1, result1)=> {
4       if (err1) callback(err1);
5       else second(result1,
6         (err2, result2)=> {
7           if (err2) callback(err2);
8           else third(result2,
9             (err3, result3)=> {
10              if (err3) callback(err3);
11              else
12                callback(null, result3);
13            }); }); });
14 }
```

Using ES6 Promises

```
1 function do123(arg){
2   return first(arg)
3     .then(second)
4     .then(third)
5 }
6
7
8
9
10
11
12
13
14
```



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
- **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



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



```
1 var action = new Promise((resolve, reject)=> {
2   var result = null;
3   // do some asynchronous stuff ...
4   if (noError) resolve(result);
5   else reject(new Error("Something Wrong"));
6   // ... do some other stuff
7 });
```

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



```
1 var action = new Promise((resolve, reject) => {  
2   setTimeout(() => {  
3     if (Math.random() > 0.5) resolve("Success!");  
4     else reject(new Error("LowValueError"));  
5   }, 1000);  
6 });  
7
```

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



```
1 action.then(  
2   (result)=> console.log(result), // result: "Success!"  
3   (error)=> console.log(error)    // error: Error("LowValueError")  
4 );  
5  
6
```

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



```
1 action.then(  
2   (result)=> console.log(result), // result: "Success!"  
3   (error)=> console.log(error)    // error: Error("LowValueError")  
4 )  
5 .then(()=> console.log("A"));  
6
```

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



```
1 action.then(  
2   (result)=> console.log(result), // result: "Success!"  
3   (error)=> console.log(error)    // error: Error("LowValueError")  
4 )  
5 .then(()=> console.log("A"))  
6 .then(()=> console.log("B"));
```

Class Activity: Promise Chaining



[lectures/lecture-9/activity1.js](https://github.com/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

```
1 function resolveAfter (time){  
2   // to implement  
3 }  
4  
5 resolveAfter(500)  
6 .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



```
1 action.then(  
2   (result)=> console.log(result), // result: "Success!"  
3   (error)=> console.log(error)    // error: Error("LowValueError")  
4 )  
5 .catch((err)=> console.log(err));  
6
```

Promise

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



```
1 action.then(  
2   (result)=> console.log(result), // result: "Success!"  
3   (error)=> console.log(error)    // error: Error("LowValueError")  
4 )  
5 .catch((err)=> console.log(err))  
6 .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



```
1 action.then(  
2   (result)=> console.log(result), // result: "Success!"  
3   (error)=> console.log(error)    // error: Error("LowValueError")  
4 )  
5 .catch((err)=> console.log(err))  
6 .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`



```
1 action.then(  
2   (result)=> {  
3     return "Action Resolved"  
4   },  
5   (error)=> {  
6     return "Action Rejected"  
7   })  
8 .then((result)=> console.log("Success: " + result),  
9       (error)=> console.log("Error: " + error.message));  
10  
11 // 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`



```
1 action.then(  
2   (result)=> {  
3     return Promise.reject("Action Resolved")  
4   },  
5   (error)=> {  
6     return Promise.resolve("Action Rejected")  
7   })  
8 .then((result)=> console.log("Success: " + result),  
9   (error)=> console.log("Error: " + error.message));  
10  
11 // 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`



```
1 action.then(  
2   (result)=> {  
3     return new Promise((resolve)=> resolve("Action Resolved"))  
4   },  
5   (error)=> {  
6     throw new Error("Action Rejected")  
7   })  
8 .then((result)=> console.log("Success: " + result),  
9   (error)=> console.log("Error: " + error.message));  
10  
11 // 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

```
1 var fs = require("fs");    // you can use fs.readFile
2
3 function readFile (filepath){
4     // to implement
5 }
6
7 readFile("example.txt")
8 .then((result)=> console.log(result.length))
9 .catch((error)=> console.log(error));
10
```

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



```
1 function signDocument(userID, fileURL){
2   return getUser(userID)
3     .then((user)=> downloadFile(fileURL, user.apiKey))
4     .then((file)=> requestNotary(file, user.cert))
5     .then((signed)=> updateRecord(userID, signed.hash))
6     .then(()=> (true), (err)=> Promise.reject(err))
7 }
8
9 var app = express();
10 app.post("/sign-request", (req, res)=> {
11   signDocument(req.session.username, req.body.fileURL)
12     .then(()=> res.status(200).send("Successful"))
13     .catch((err)=> res.status(500).send("Server Error"))
14 });
```

Promise

- Using the static function `Promise.all`, we can wait for multiple concurrent `Promises` 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)



```
1 var multi = Promise.all([
2   new Promise((resolve)=> setTimeout(()=> resolve("A"), 2000)),
3   new Promise((resolve)=> setTimeout(()=> resolve("B"), 3000)),
4   new Promise((resolve)=> setTimeout(()=> resolve("C"), 1000)),
5 ]);
6
7 multi.then(
8   (results)=> console.log(results),
9   (error)=> console.log(error));
10
```

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



```
1 var multi = Promise.race([
2   new Promise((resolve)=> setTimeout(()=> resolve("A"), 2000)),
3   new Promise((resolve)=> setTimeout(()=> resolve("B"), 3000)),
4   new Promise((resolve)=> setTimeout(()=> resolve("C"), 1000)),
5 ]);
6
7 multi.then(
8   (result)=> console.log(result),
9   (error)=> console.log(error));
10
```

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.

