Lecture 10.2

Promises

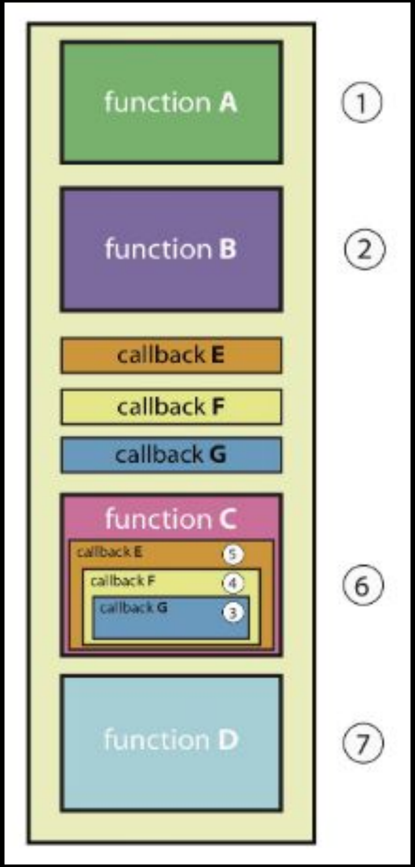
Topics

- Synchronous vs Asynchronous
- ES6 - Promises
  - Promise Methods
  - Callbacks vs Promises

Synchronous vs Asynchronous

JavaScript is Synchronous!

- JavaScript is a synchronous, blocking, single-threaded language.
- That just means that only one operation can be in progress at a time.
- JavaScript can behave in an asynchronous way, but it's not built-in. We can use Asynchronous Callbacks and Promises



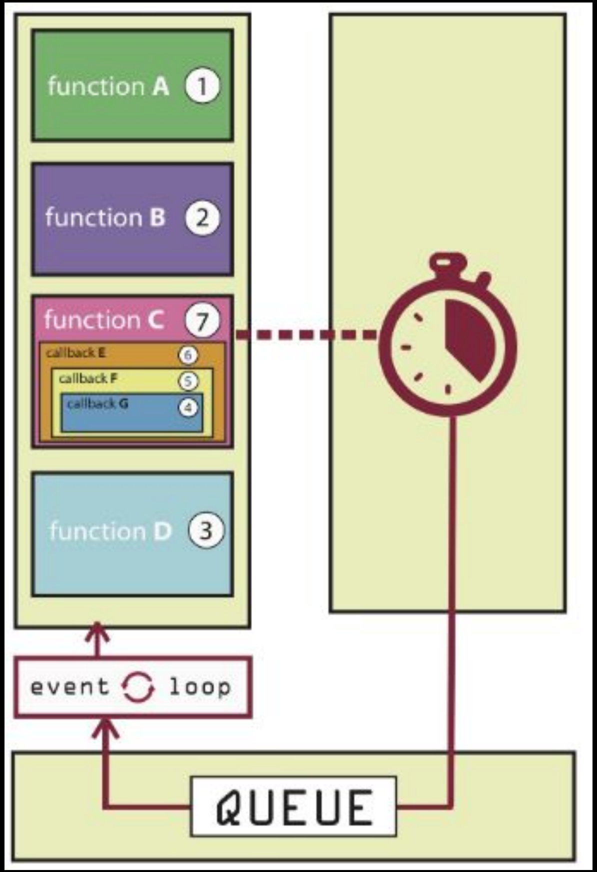
(Blocking) Synchronous Stack

Asynchronous Callbacks

- The earliest and most straightforward solution to being stuck in the synchronous world is using asynchronous callbacks (ie. setTimeout())

- This is a great solution, but you can't predict exactly when function C will resolve, you have to nest all dependent functions within it.

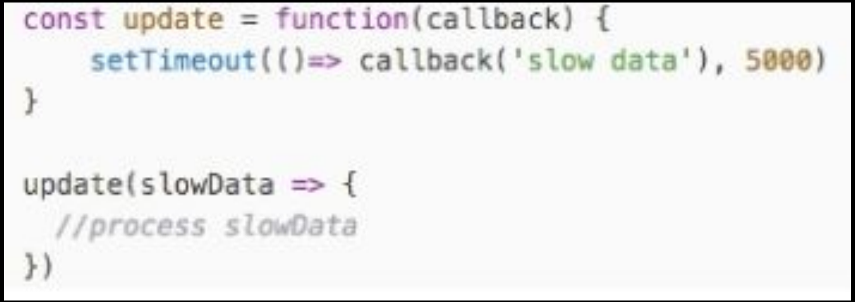
- This gets messy fast and leads to the infamous callback hell that no one wants to deal with. *(Promises solved this)*



Asynchronous Programming

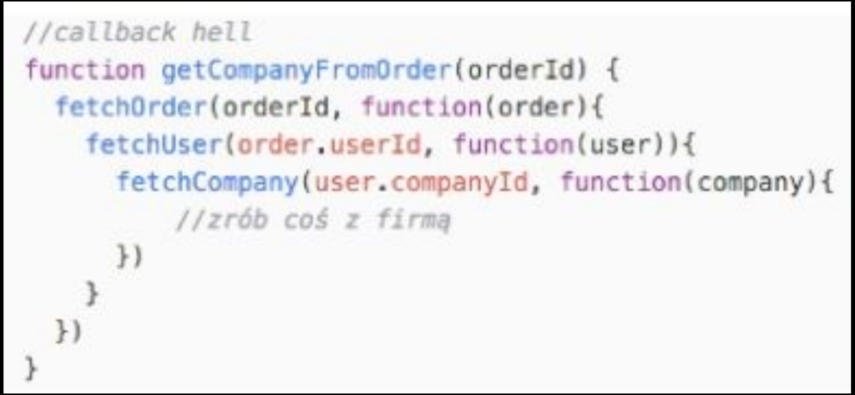
- Async programming is common in JavaScript (animations, server requests, etc.)
- The classic solution is the callback
- The main problem is that there is only one callback per async task
- Another problem is that nested callback functions create messy code.

One callback function only



```
const update = function(callback) {
    setTimeout(()=> callback('slow data'), 5000)
}

update(slowData => {
  //process slowData
})
```

Nested callback hell

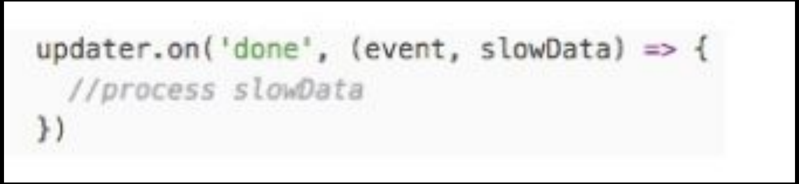


```
//callback hell
function getCompanyFromOrder(orderId) {
  fetchOrder(orderId, function(order){
    fetchUser(order.userId, function(user)){
      fetchCompany(user.companyId, function(company){
          //zrób coś z firmą
      })
    }
  })
}
```

ES5: Async Programming

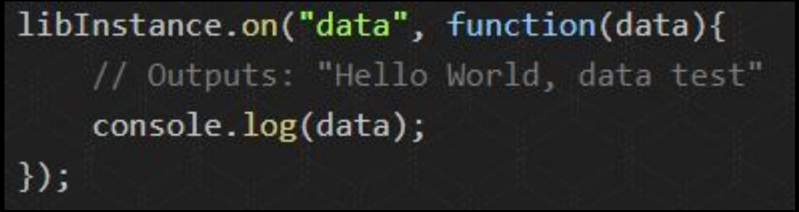
- Second try with async programming in ES5 was Listeners
- The problem with Listeners is there is no reaction when an async function ends before the listener registers. It is often hard to debug

Async function can end before listener is registered



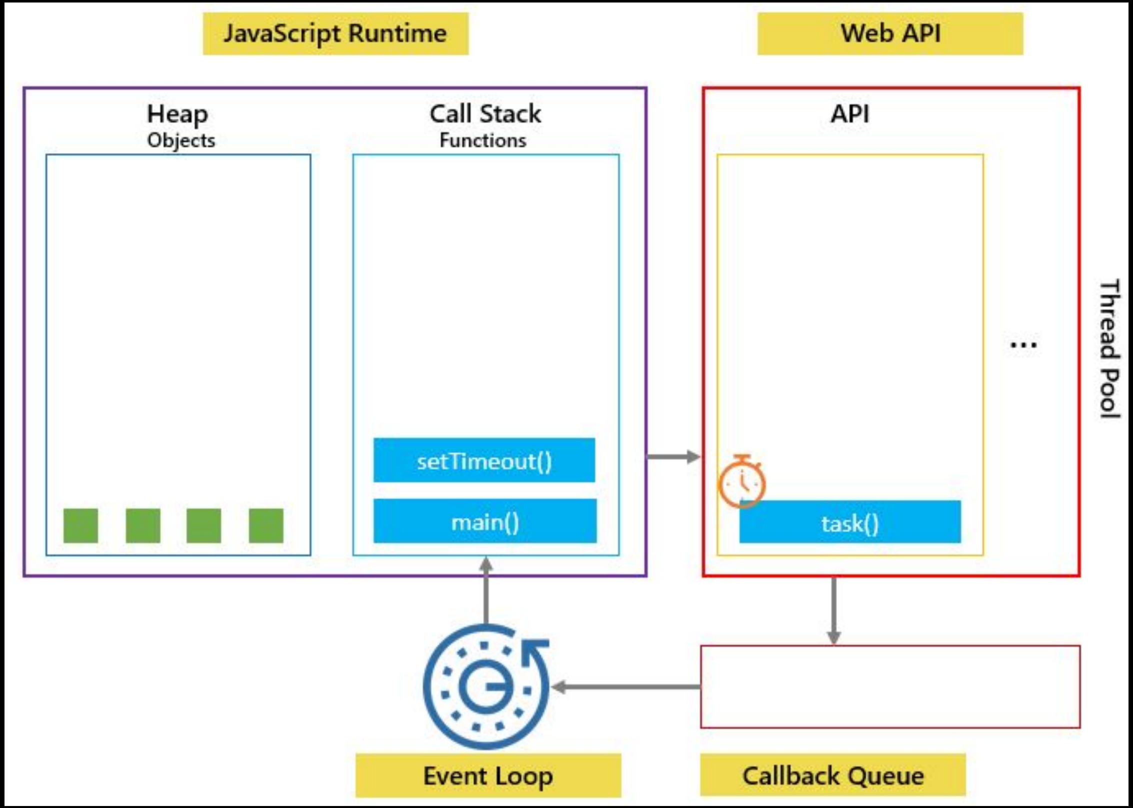
```
updater.on('done', (event, slowData) => {
  //process slowData
})
```

Add the "data" event listener and add some callback



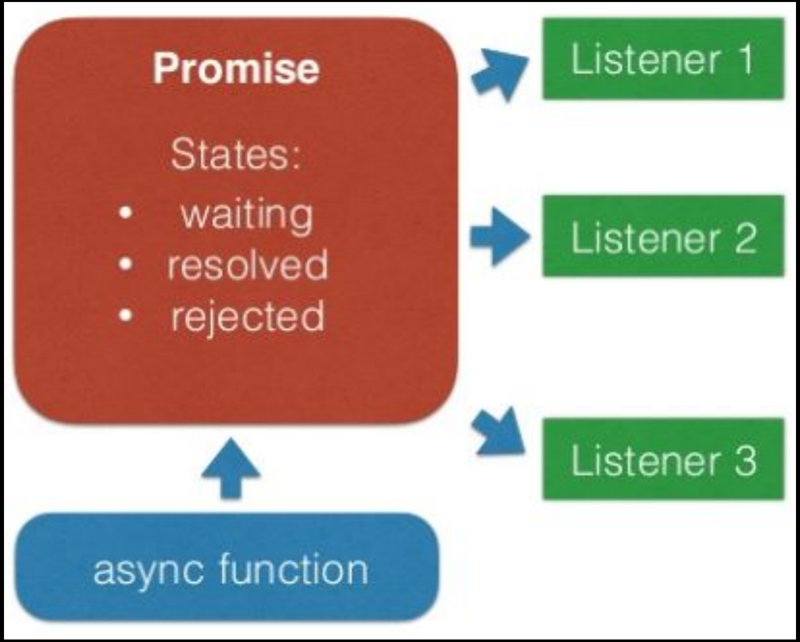
```
libInstance.on("data", function(data){
    // Outputs: "Hello World, data test"
    console.log(data);
});
```

JavaScript Runtime



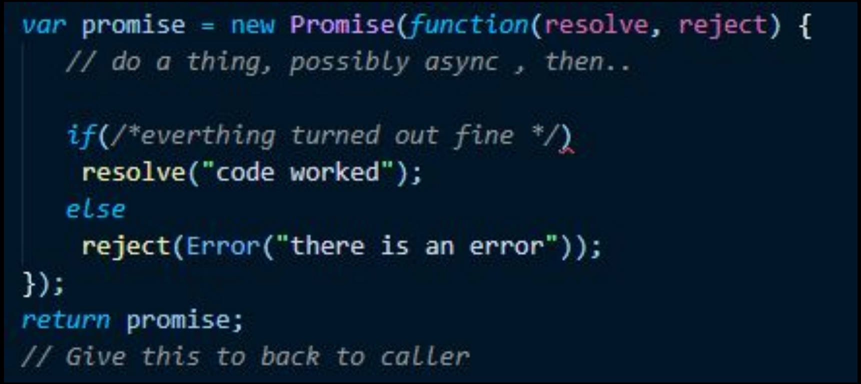
Promises

ES6 **Promise**

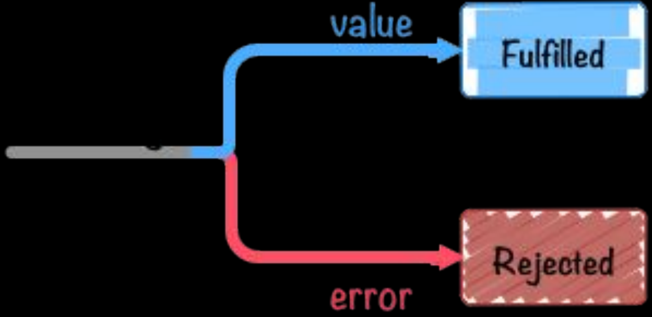


- A Promise is an object that keeps a result of an async function (waiting, resolved, rejected)
- Fixes earlier problem with listeners, since callback is called even if the async function completed the task earlier
- <u>Note: waiting state is also, often referred to as the pending state</u>

ES6 Promise



```
var promise = new Promise(function(resolve, reject) {
   // do a thing, possibly async , then..

   if(/*everthing turned out fine */)
    resolve("code worked");
   else
    reject(Error("there is an error"));
});
return promise;
// Give this to back to caller
```

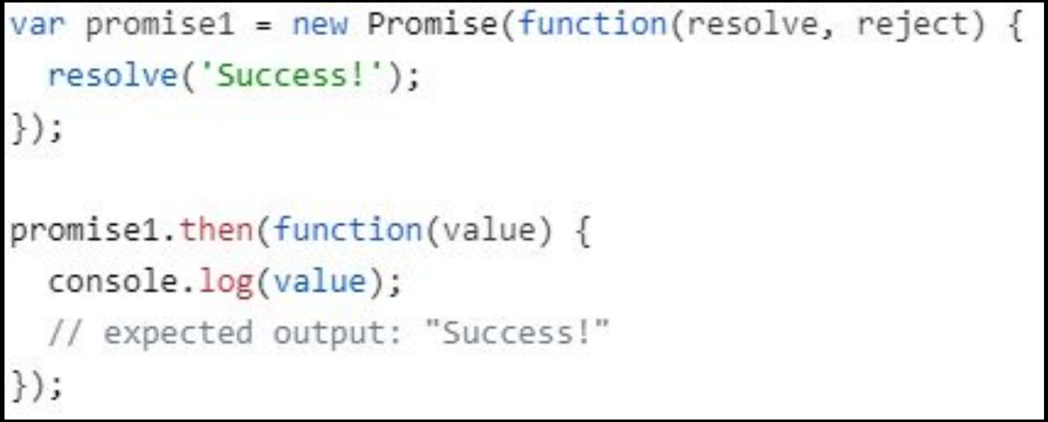


- The Promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.

- First we need to declare a new Promise object
- Second we need to pass into the constructor two parameters
  - The callback function for the successful completion of the work
  - The callback function to handle the case when an error occurs

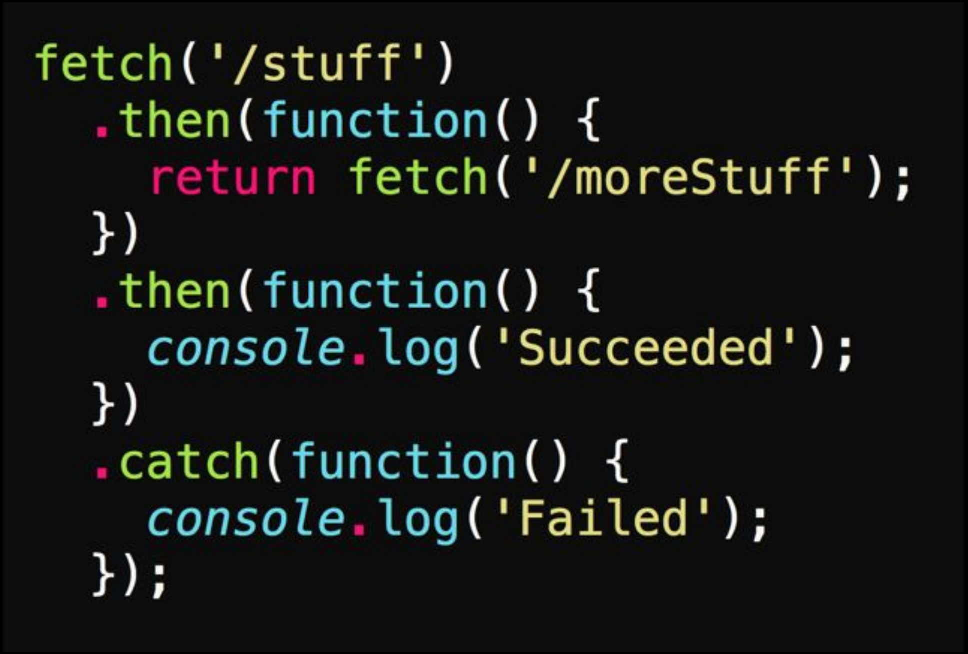
Promise.prototype.then()

- The then() method returns a Promise. It takes up to two arguments: callback functions for the success and failure cases of the Promise.



```
var promise1 = new Promise(function(resolve, reject) {
  resolve('Success!');
});

promise1.then(function(value) {
  console.log(value);
  // expected output: "Success!"
});
```

Promise Chaining

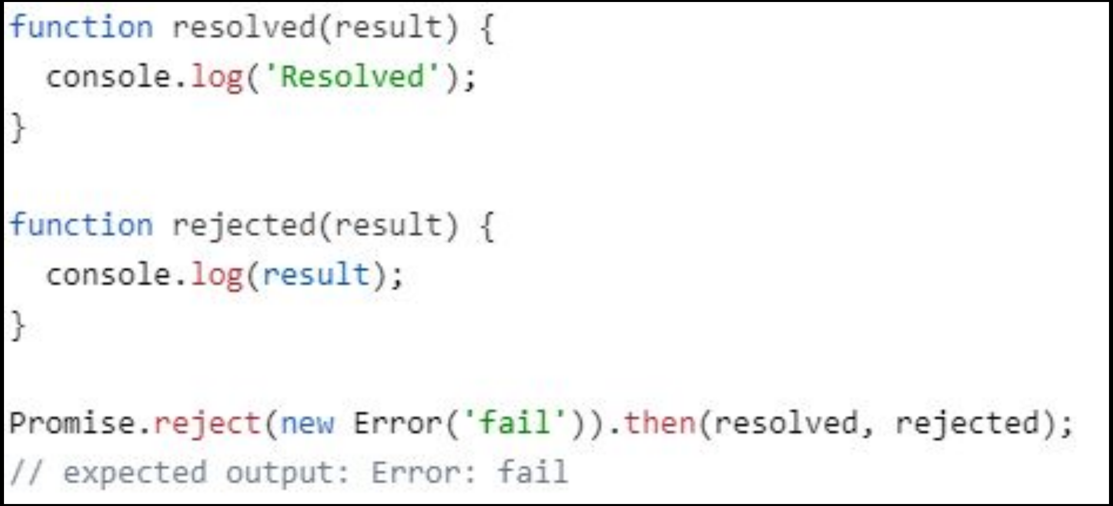


```
fetch('/stuff')
  .then(function() {
    return fetch('/moreStuff');
  })
  .then(function() {
    console.log('Succeeded');
  })
  .catch(function() {
    console.log('Failed');
  });
```

- The then() method returns a Promise which allow for chaining.
- The callback function in then can return a resolve or reject.

Promise.reject()

- The Promise.reject() method returns a Promise object that is rejected with a given reason.

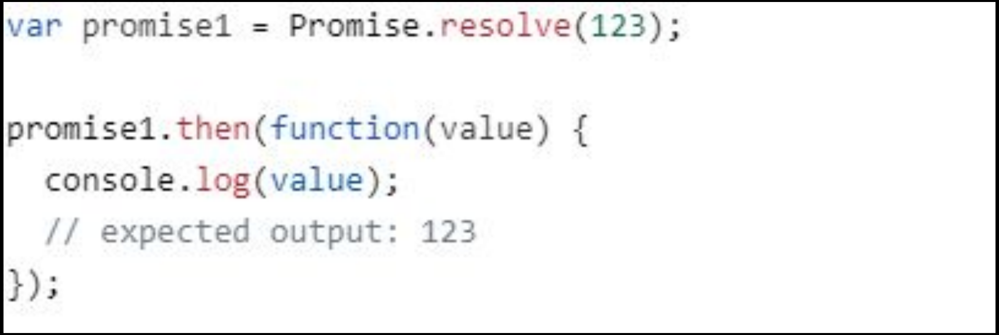


```
function resolved(result) {
  console.log('Resolved');
}

function rejected(result) {
  console.log(result);
}

Promise.reject(new Error('fail')).then(resolved, rejected);
// expected output: Error: fail
```

Promise.resolve()

- The Promise.resolve() method returns a Promise object that is resolved with a given value.
    - If the value is a promise, that promise is returned;
    - if the value is a thenable (i.e. has a "then" method), the returned promise will "follow" that thenable, adopting its eventual state; otherwise the returned promise will be fulfilled with the value. .

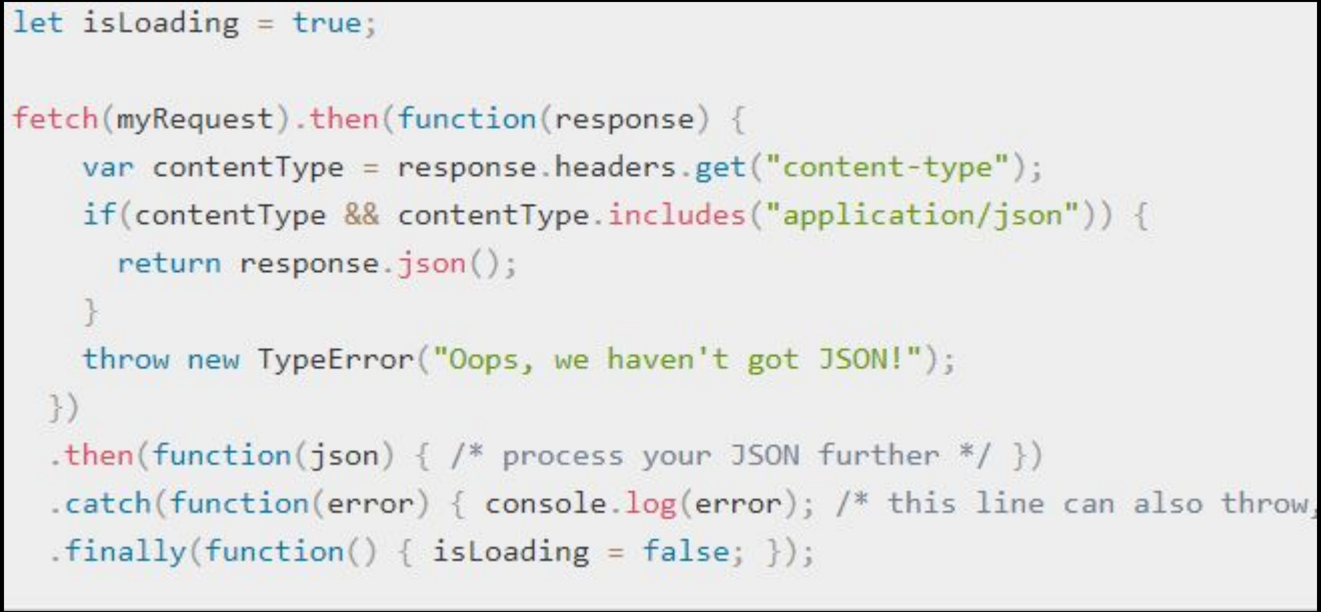
- Warning: Do not call Promise.resolve on a thenable that resolves to itself. This will cause infinite recursion as it tries to flatten what seems to be an infinitely nested promise.



```
var promise1 = Promise.resolve(123);

promise1.then(function(value) {
  console.log(value);
  // expected output: 123
});
```

Promise.prototype.catch()...and finally()

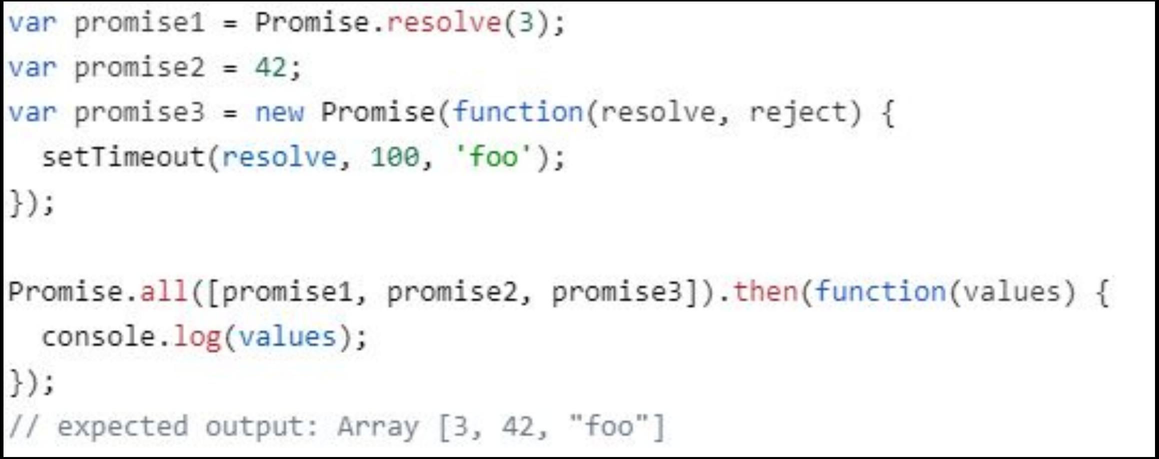
- The catch() method returns a Promise and deals with rejected cases only.
- The finally() method returns a Promise. When the promise is settled, i.e either fulfilled or rejected, the specified callback function is executed.



```
let isLoading = true;

fetch(myRequest).then(function(response) {
    var contentType = response.headers.get("content-type");
    if(contentType && contentType.includes("application/json")) {
      return response.json();
    }
    throw new TypeError("Oops, we haven't got JSON!");
  })
  .then(function(json) { /* process your JSON further */ })
  .catch(function(error) { console.log(error); /* this line can also throw
  .finally(function() { isLoading = false; });
```

Promise.all()

- The Promise.all() method returns a single Promise that resolves when all of the promises passed as an iterable have resolved or when the iterable contains no promises.

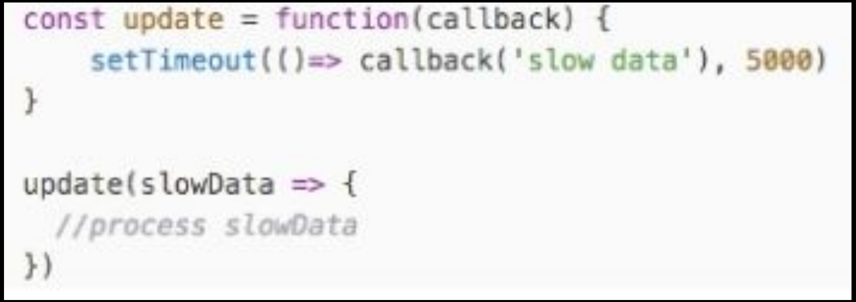


```
var promise1 = Promise.resolve(3);
var promise2 = 42;
var promise3 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([promise1, promise2, promise3]).then(function(values) {
  console.log(values);
});
// expected output: Array [3, 42, "foo"]
```

Callback vs Promises

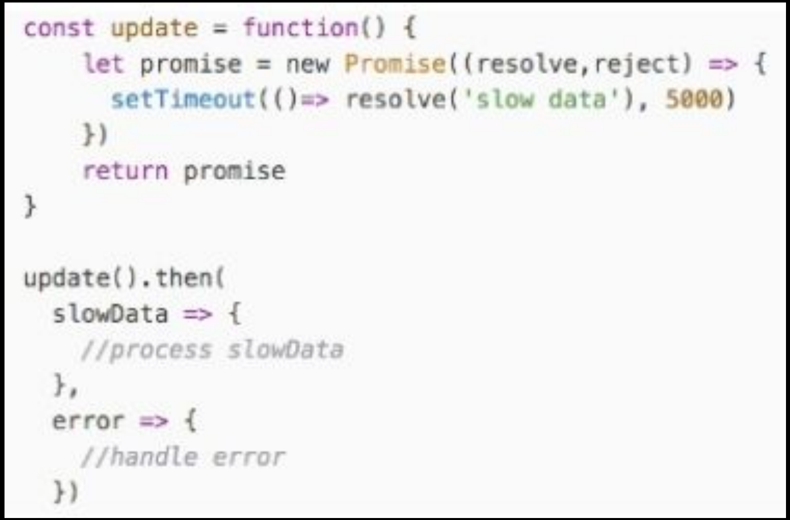
Callback vs Promise

One callback function only



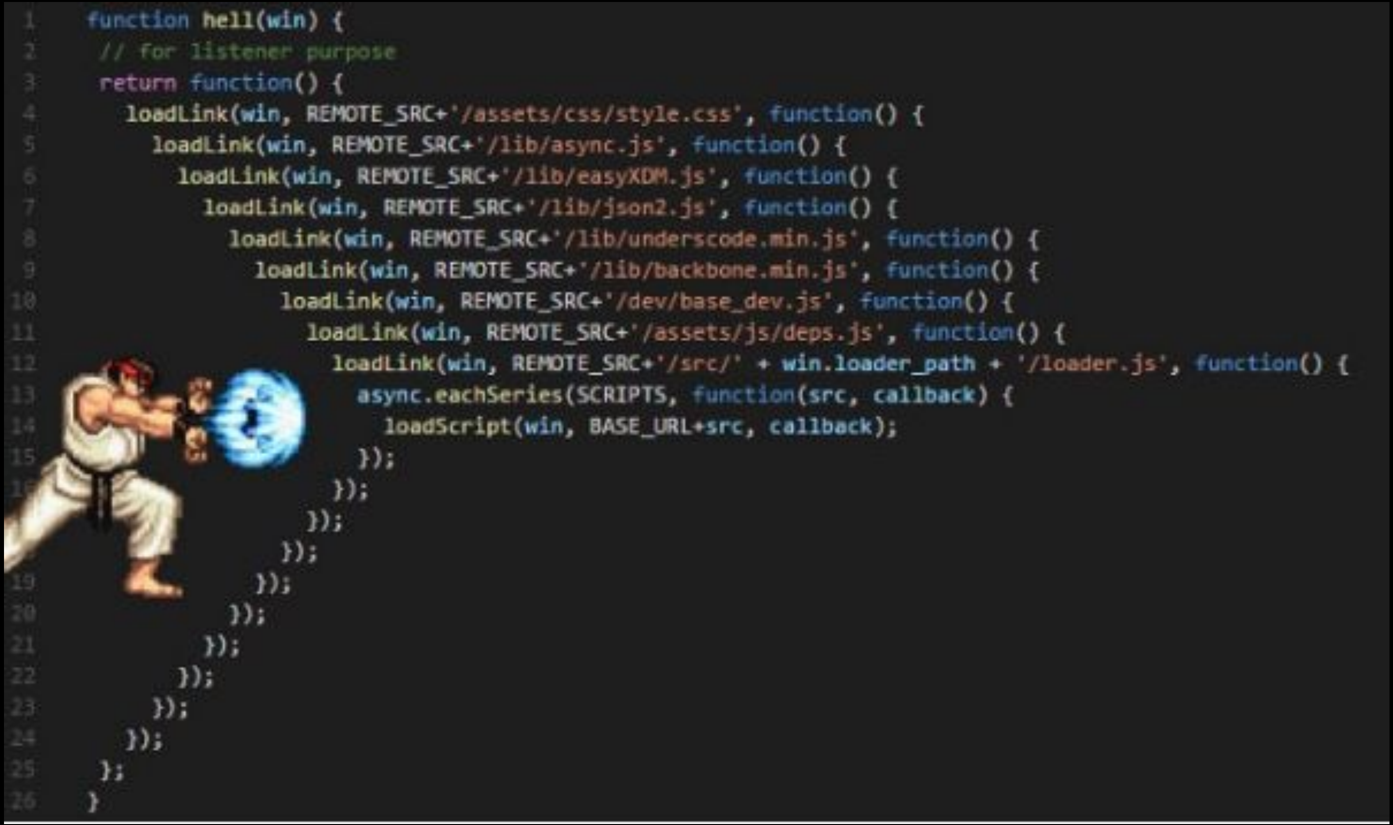
```
const update = function(callback) {
    setTimeout(()=> callback('slow data'), 5000)
}

update(slowData => {
  //process slowData
})
```

Same method using resolving a Promise



```
const update = function() {
    let promise = new Promise((resolve,reject) => {
      setTimeout(()=> resolve('slow data'), 5000)
    })
    return promise
}

update().then(
  slowData => {
    //process slowData
  },
  error => {
    //handle error
  })
```

Callback Hell Revisited

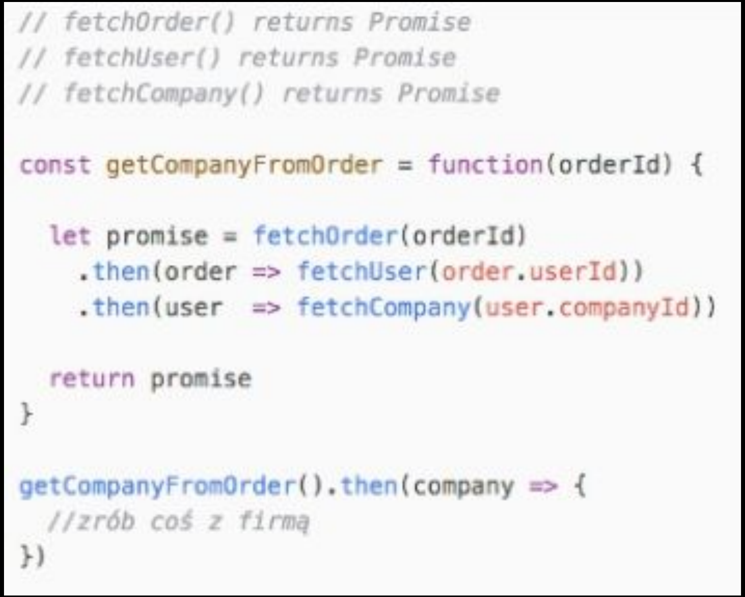


```
function hell(win) {
 // for listener purpose
 return function() {
   loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
     loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
       loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
         loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
           loadLink(win, REMOTE_SRC+'/lib/underscode.min.js', function() {
             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
                     async.eachSeries(SCRIPTS, function(src, callback) {
                       loadScript(win, BASE_URL+src, callback);
                     });
                   });
                 });
               });
             });
           });
         });
       });
     });
   });
 };
}
```

Nested Callbacks vs Promises

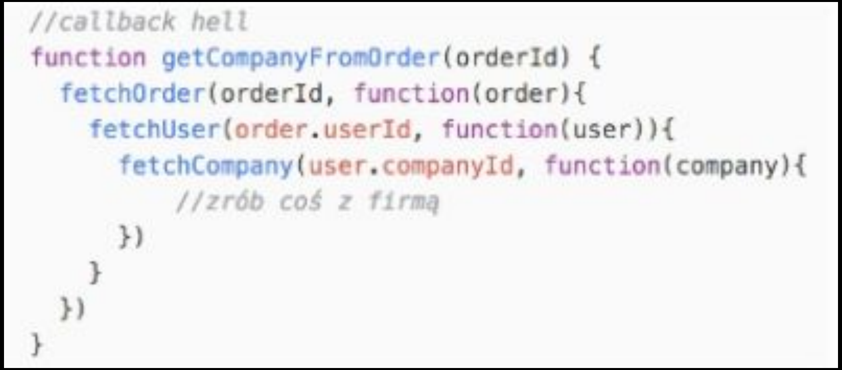
- A main advantage is nested promises can be flattened to avoid "callback hell"

ES6 Promises



```
// fetchOrder() returns Promise
// fetchUser() returns Promise
// fetchCompany() returns Promise

const getCompanyFromOrder = function(orderId) {

  let promise = fetchOrder(orderId)
    .then(order => fetchUser(order.userId))
    .then(user  => fetchCompany(user.companyId))

  return promise
}

getCompanyFromOrder().then(company => {
  //zrób coś z firmą
})
```

ES5 Callback hell



```
//callback hell
function getCompanyFromOrder(orderId) {
  fetchOrder(orderId, function(order){
    fetchUser(order.userId, function(user)){
      fetchCompany(user.companyId, function(company){
          //zrób coś z firmą
      })
    }
  })
}
```