ES6

ES6 - Async/Await

2023 Fall

week 04 - class 09

Topics

- ES6 Features
  - Promises & Async Await

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

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

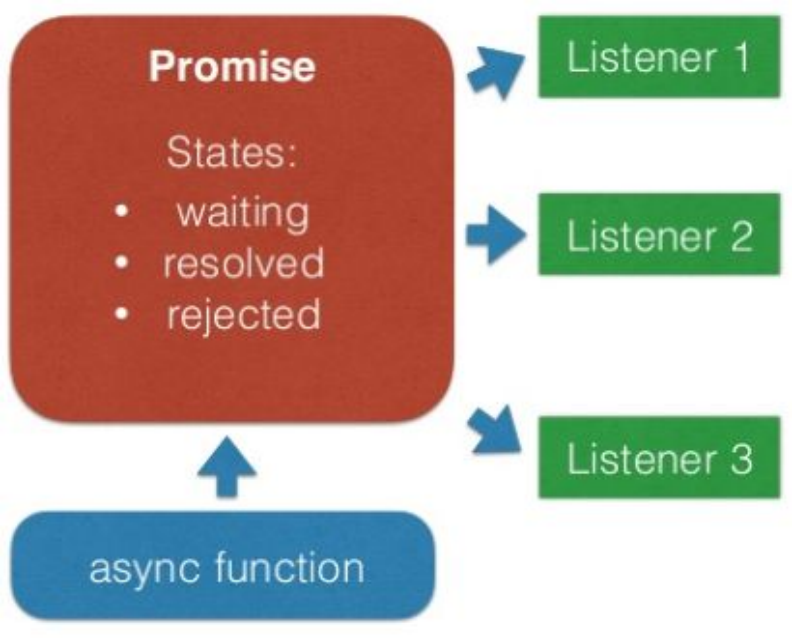
```
const update = function(callback) {
    setTimeout(()=> callback('slow data'), 5000)
}

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
      })
    }
  })
}
```

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

Async/Await

ES6 async/await

- Async/await is a new way to write asynchronous code. Previous options for asynchronous code are callbacks and promises.
- Async/await is actually built on top of promises. It cannot be used with plain callbacks or node callbacks.
- Async/await is, like promises, non blocking.
- Async/await makes asynchronous code look and behave a little more like synchronous code.

async/await syntax

- Any async function returns a promise implicitly and the resolve value of the promise will be whatever you return from the function (which is string "done" in our example)
- In the example, await getJSON() means that the console.log call will wait until getJSON() promise resolves and print the value

Handling Promise

```
const makeRequest = () =>
  getJSON()
    .then(data => {
      console.log(data)
      return "done"
    })

makeRequest()
```

async/await

```
const makeRequest = async () => {
  console.log(await getJSON())
  return "done"
}

makeRequest()
```

async/await benefits - error handling

- Error Handling with async/await makes it finally possible to handle both synchronous and asynchronous errors with the good old try/catch

try/catch on the entire Promise

```
const makeRequest = () => {
  try {
    getJSON()
      .then(result => {
        // this parse may fail
        const data = JSON.parse(result)
        console.log(data)
      })
  } catch (err) {
    console.log(err)
  }
}
```

catch will now handle parse error

```
const makeRequest = async () => {
  try {
    // this parse may fail
    const data = JSON.parse(await getJSON())
    console.log(data)
  } catch (err) {
    console.log(err)
  }
}
```

async/await benefits - cleaner code

- Avoid Nesting and write cleaner code with conditionals and async/await

Confusing conditions with nested Promises

```
const makeRequest = () => {
  return getJSON()
    .then(data => {
      if (data.needsAnotherRequest) {
        return makeAnotherRequest(data)
          .then(moreData => {
            console.log(moreData)
            return moreData
          })
      } else {
        console.log(data)
        return data
      }
    })
```

More readable when written with async/await

```
const makeRequest = async () => {
  const data = await getJSON()
  if (data.needsAnotherRequest) {
    const moreData = await makeAnotherRequest(data);
    console.log(moreData)
    return moreData
  } else {
    console.log(data)
    return data
  }
}
```

async/await benefits - intermediate values

- Intermediate values - sometimes you have a situation where you call promise1 and then use it what it returns to call promise2, then you use the results of both promises to call promise3

```
const makeRequest = () => {
  return promise1()
    .then(value1 => {
      // do something
      return promise2(value1)
        .then(value2 => {
          // do something
          return promise3(value1, value2)
        })
    })
}
```

```
const makeRequest = async () => {
  const value1 = await promise1()
  const value2 = await promise2(value1)
  return promise3(value1, value2)
}
```

async/await benefits - debugging

- Debugging promises has always been a pain for 2 reasons.
  - you can't set breakpoints in arrow functions that return expressions
  - the debugger won't step through .then blocks, it will skip because it only steps through synchronous code

Try setting a breakpoint here in nested promises

```
const makeRequest = () => {
  return callAPromise()
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => callAPromise())
}
```

await/async breakpoints

```
5   const makeRequest = async () => {
6     await callAPromise()
7     await callAPromise()
8     await callAPromise()
9     await callAPromise()
10    await callAPromise()
11  }
```