CS 4350: Fundamentals of Software Engineering
CS 5500: Foundations of Software Engineering

Lesson 4.2: Writing functions with async/await

Jon Bell, John Boyland, Mitch Wand

Khoury College of Computer Sciences

Learning Objectives for this Lesson

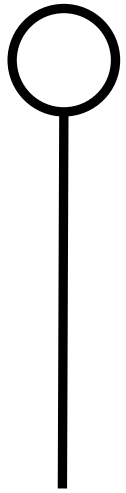
- By the end of this lesson you should be prepared to:
  - Explain how a sequence of then/catch handlers handle successful promises and errors
  - Explain how async/await works with try/catch to make asynchronous programming easier

Outline of this Lesson

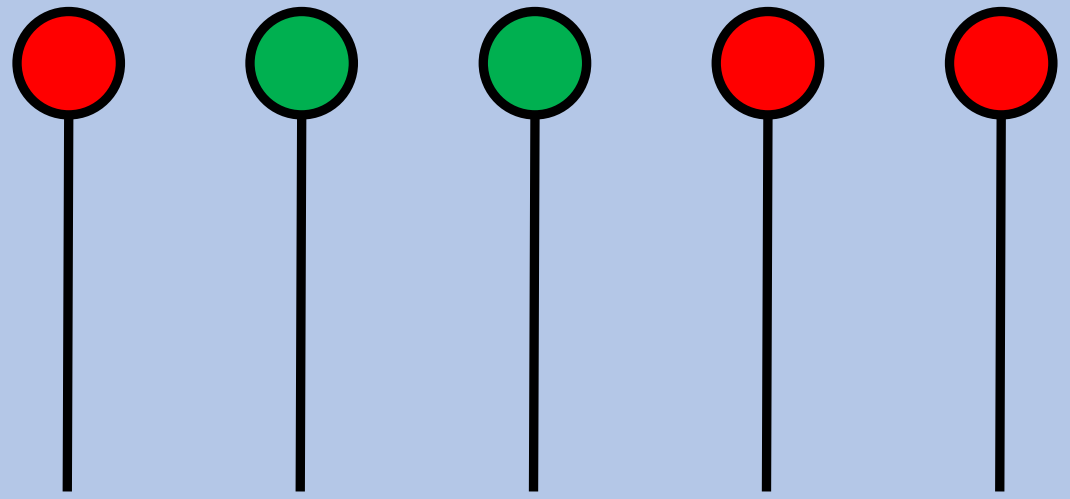
- What happens when a promise is rejected?
- Creating sequences of actions by writing chains of .**then** and .**catch** blocks
- Using **async** and **await** to avoid writing these chains.

Review: The Javascript runtime maintains a pool of handlers.

The running handler

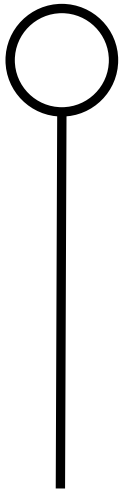


The Pool of Waiting handlers

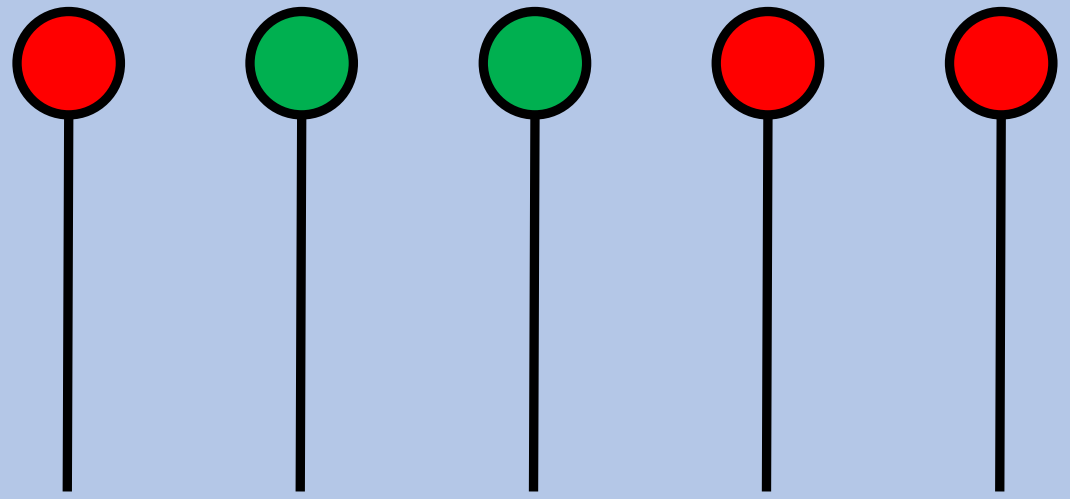


When the running event handler completes, the scheduler chooses one of the other ready event handlers to execute

The running event handler

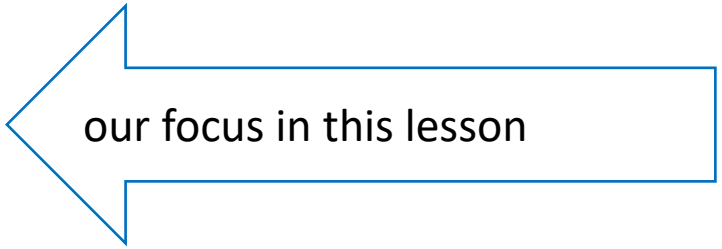


The Pool of Waiting event handlers



How can a handler become ready?

- There are roughly 3 ways in which a handler can become ready:
  - it can become ready at a specific time.
  - it can become ready when some input/output event occurs
  - it can become ready when some other handler or handlers complete.



You will most likely not be building promises from scratch

- Asynchronous operations (like input/output operations) are typically exported as promises (or as functions that return promises)
- So we'll concentrate on using promises, by using .**then** and .**catch** properties.
- For our examples, we'll create promises using a function with the following interface:

```
function makePromise1(promiseName: string, shouldSucceed: boolean, value?: number)
    : Promise<number>
    // returns a promise that fulfills with the given value if shouldSucceed is true
    // and that is rejected with the string "promiseName was rejected"
    // otherwise.   'value' is an optional argument when shouldSucceed is false
```

examples-4.2/example1.ts

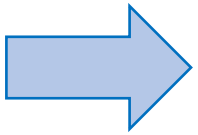
What happens if a promise fails?

```
import makePromise1 from './promiseMaker'

console.log("main handler starting")

makePromise1("promise1",true, 10)
 .then(n => console.log(`promise1 passed ${n}
 to its successor`))
makePromise1("promise2",false)
 .then(n => console.log(`promise2 passed ${n}
 to its successor`))
console.log('main handler finished')
```

Sorry for the bad line breaks. Gotta fit it on the slide ☺



```
main handler starting
creating new promise promise1
creating new promise promise2
main handler finished
promise promise1 now running; flag =
true
promise promise1 now fulfilling with
10
promise1 passed 10 to its successor
promise promise2 now running; flag =
false
promise promise2 now rejecting
(node:19860)
UnhandledPromiseRejectionWarning:
promise promise2 was rejected
(node:19860)
UnhandledPromiseRejectionWarning:
Unhandled promise rejection.
```

What happened here?

- .then handlers only handle promises that succeed
- To handle failure, you need a .catch() handler

examples-4.2/example2.ts

...with a .catch() handler

```
import makePromise1 from './promiseMaker'

console.log("main handler starting")

makePromise1("promise1",true, 10)
 .then(n => console.log(`promise1 passed ${n}
to its successor`))
makePromise1("promise2",false)
 .catch(n => console.log(`promise2 passed ${n}
 to its successor`))

 console.log('main handler finished')
```

'n' is bound to the rejection message produced by the promise, in this case, "promise2 was rejected."

```
main handler starting
creating new promise promise1
creating new promise promise2
main thread finished
promise promise1 now running; flag =
true
promise promise1 now fulfilling with 10
promise1 passed 10 to its successor
promise promise2 now running; flag =
false
promise promise2 now rejecting
promise2 passed promise promise2 was
rejected to its successor
```

.then() and .catch() blocks can themselves succeed or fail

- throwing an error counts as failure
- anything else counts as succeeding
- This determines which then/catch blocks get executed.

.then and .catch blocks can pass values to their successors using **return**

examples-4.2/example3.ts

```
import makePromise1 from './promiseMaker'

console.log("main handler starting")

function driver(promiseName: string, flag: boolean) {
    console.log(`starting driver(${flag})`)
    makePromise1(promiseName,flag,10)
        .then(n => {console.log(`promise ${promiseName} fulfil
led and passed ${n} to its successor`);
                    return n+1
                })
        .then(m => console.log(`the second then block received
 ${m}`))
        .catch(n => console.log(`promise ${promiseName} reject
ed and passed "${n}" to its successor`))
}

driver("promise1",true)
driver("promise2",false)

console.log('main handler finished')
```

This works inside either a then() or a catch() block

```
main handler starting
starting driver(true)
creating new promise promise1
starting driver(false)
creating new promise promise2
main handler finished
promise promise1 now running;
flag = true
promise promise1 now
fulfilling with 10
promise promise1 fulfilled and
passed 10 to its successor
the second then block received
11
promise promise2 now running;
flag = false
promise promise2 now rejecting
promise promise2 rejected and
passed "promise promise2 was
rejected" to its successor
```

.then and .catch blocks can also throw errors to their successors

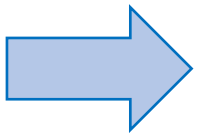
examples-4.2/example4.ts

```
import makePromise1 from './promiseMaker'

console.log("main handler starting")

function driver(promiseName: string, flag: boolean) {
    console.log(`starting driver(${promiseName})`)
    makePromise1(promiseName, flag, 10)
        .then(n => {
            console.log(`promise ${promiseName} fulfilled and pa
ssed ${n} to its successor`);
            console.log(`the then block of ${promiseName} will n
ow throw an error`);
            throw new Error("my error 1")
        })
        .then(m => console.log(`the second then block received $
{m}`))
        .catch(n => console.log(`promise ${promiseName} rejected
 and passed "${n}" to its successor`))
}


driver("promise1", true)
driver("promise2", false)
console.log('main handler finished')
```



```
main handler starting
starting driver(promise1)
creating new promise promise1
starting driver(promise2)
creating new promise promise2
main handler finished
promise promise1 now running; flag =
true
promise promise1 now fulfilling with 10
promise promise1 fulfilled and passed 10
to its successor
the then block of promise1 will now
throw an error
promise promise1 rejected and passed
"Error: my error 1" to its successor
promise promise2 now running; flag =
false
promise promise2 now rejecting
promise promise2 rejected and passed
"promise promise2 was rejected" to its
successor
```

Chained .then and .catch blocks

- This leads to code like this:

```
somePromise
  .then()
  .then()
  .then()
  .catch()
  .then()          // if there's more to do after the catch
  .then()
  .catch()
```

- and what if there are conditionals to worry about?

Yuck!

Avoiding this with async/await

- An async function is declared with the **async** keyword.
- Within an async function, you can call another promise function, and **await** its result.
- You can also use try/catch within the body of the async function; the catch block in the try/catch becomes a catch handler on the async function you just called.
- This sounds more complicated than it is. Let's go back a few steps.

Here's the pattern

examples-4.2/example5.ts

instead of this

```
function f() {
    doThisNow()
    promiseReturningFunction()
        .then(value =>onSuccess(value))
        .catch(errmsg => onFailure(errmsg))
}
```

we do this right now, in the caller's handler

we'll do this stuff in the new handler, sometime after the caller's handler is finished

write this:

```
async function f {
    try {
        doThisNow()
        const value = await promiseReturningFunction()
        onSuccess(value)
    } catch (errmsg) {
        onFailure(errmsg)
    }
}
```

we do this right now, in the caller's handler

we'll do this stuff in the new handler, sometime after the caller's handler is finished

Here's the pattern (2)

examples-4.2/example5.ts

```
function f() {
    doThisNow()
    promiseReturningFunction()
        .then(value =>onSuccess(value))
        .catch(errmsg => onFailure(errmsg))
}
```

into this

the **async** keyword tells the system to translate

```
async function f {
    try {
        doThisNow()
        const value = await promiseReturningFunction()
        onSuccess(value)
    } catch (errmsg) {
        onFailure(errmsg)
    }
}
```

this

we do this right now, in the caller's handler

we'll do this stuff in the new handler, sometime after the caller's handler is finished

we do this right now, in the caller's handler

we'll do this stuff in the new handler, sometime after the caller's handler is finished

Here's an example (original)

examples-4.2/example6.ts

```
function driver(promiseName: string, flag: boolean) {
    console.log(`starting driver(${flag})`)
    makePromise1(promiseName,flag,10)
        .then(n => {console.log(`promise ${promiseName} fulfilled and passed ${n}
 to its successor`);
                    return n+1
                })
        .then(m => console.log(`the second then block received ${m}`))
        .catch(n => console.log(`promise ${promiseName} rejected and passed "${n}
" to its successor`))
}
```

examples-4.2/example6.ts

Example rewritten with async/await

```
async function driver2(promiseName: string, flag: boolean) {
    try {
        console.log(`starting driver2(${flag})`)
        const n = await makePromise1(promiseName, flag, 10)
        console.log(`promise ${promiseName} fulfilled and passed ${n} to its suc
cessor`);
        const m = n + 1
        console.log(`the second then block received ${m}`)
    } catch (n) { console.log(`promise ${promiseName} rejected and passed "${n}"
 to its successor`) }
}
```

examples-4.2/example6.ts

Let's run them both and compare (1)

```
import makePromise1 from './promiseMaker'

console.log("main handler starting")

function driver(promiseName: string, flag:

async function driver2(promiseName: string
 {...}

console.log("first group")
driver("promise1",true)
driver("promise2",false)
driver2("promise1a",true)
driver2("promise2a",false)

console.log('main handler finished')
```

```
main handler starting
first group
starting driver(true)
creating new promise promise1
starting driver(false)
creating new promise promise2
starting driver2(true)
creating new promise promise1a
starting driver2(false)
creating new promise promise2a
main handler finished
promise promise1 now running; flag = true
promise promise1 now fulfilling with 10
promise promise1 fulfilled and passed 10 to its
successor
the second then block received 11
promise promise2 now running; flag = false
promise promise2 now rejecting
promise promise2 rejected and passed "promise
promise2 was rejected" to its successor
(continued on next slide)
```

examples-4.2/example6.ts

Let's run them both and compare (2)

```
import makePromise1 from './promiseMaker'

console.log("main handler starting")

function driver(promiseName: string, flag: b

async function driver2(promiseName: string,
 {...}

console.log("first group")
driver("promise1",true)
driver("promise2",false)
driver2("promise1a",true)
driver2("promise2a",false)

console.log('main handler finished')
```

```
(continued from preceding slide)
promise promise1a now running; flag = true
promise promise1a now fulfilling with 10
promise promise1a fulfilled and passed 10 to
its successor
the second then block received 11
promise promise2a now running; flag = false
promise promise2a now rejecting
promise promise2a rejected and passed "promise
promise2a was rejected" to its successor
```

Same behavior!

The outputs, side by side

```
main handler starting
first group
starting driver(true)
creating new promise promise1
starting driver(false)
creating new promise promise2
starting driver2(true)
creating new promise promise1a
starting driver2(false)
creating new promise promise2a
main handler finished
promise promise1 now running; flag = true
promise promise1 now fulfilling with 10
promise promise1 fulfilled and passed 10 to its
successor
the second then block received 11
promise promise2 now running; flag = false
promise promise2 now rejecting
promise promise2 rejected and passed "promise
promise2 was rejected" to its successor
(continued on next slide)
```

Indeed, same behavior!

```
(continued from preceding slide)
promise promise1a now running; flag = true
promise promise1a now fulfilling with 10
promise promise1a fulfilled and passed 10 to
its successor
the second then block received 11
promise promise2a now running; flag = false
promise promise2a now rejecting
promise promise2a rejected and passed "promise
promise2a was rejected" to its successor
```

Things to know about **async/await**

- An async function always returns a promise.
- Because a promise is created, it is automatically thrown in the pool of handlers to be run when ready
- The async keyword tells the compiler to do the translation
- Therefore, await makes no sense except in the body of an async function.
- The try/catch is optional.

That was a long story to reach a simple conclusion

- A useful but complex pattern of behaviors is encapsulated in a single language construct.
- In the olden days, this might have been a "design pattern"
- Illustrates the power of programming-language technology

Review: Learning Objectives for this Lesson

- You should now be able to:
  - Explain how a sequence of then/catch handlers handle successful promises and errors
  - Explain how async/await works with try/catch to make asynchronous programming easier

Next Steps

- Be prepared to explain each line in the output examples
- Create some examples like the ones here and try to predict what they will do.
- Think of some good questions to bring to class!