Functional Programming (FP) 3

Async: Promises, async and errors

Overview

- Error handling
- Composing Promises
- Common mistakes
- Concurrent Promises
- Async / Await
- Try / Catch
- Exercises

Objectives

- Understand how to handle errors
- Understand how to compose promises
- Know some of the common mistakes
- Know how to use `async/await` keywords

Catching Errors

When using promises, we need to catch errors slightly differently than normal.
We can use `Promise.catch` to pass the reason a promise has been rejected to a callback function:

```
promise1.then(nextFunction).catch(errorHandler)
```

This is the standard pattern

- Any rejection or error in the chain (in `promise1` or `nextFunction`) will jump us to the error handler

Catching Errors

The `catch` only operates on the chain before it. Here, node will always call `nextFunction` whether or not there is an error in `promise1`:

```
promise1.catch(errorHandler).then(nextFunction)
```

Watch out! `nextFunction` may get no data as catch() returns undefined!

Task - promise errors

> Let's have a look at file `examples/consuming-promise-errors.ts` together

- Open the file, run it, watch how the logs appear
- Try both method chains in the file

Emoji Check:

Do you understand how to use `.catch()` in a chain to handle errors?

1. 😢 Haven't a clue, please help!
2. 🙁 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

Promises and data

Promises are usually handling data for us, not just strings or numbers.
Typically, a database or api call will return json that we will end up passing around and processing.

Task - promise data

> Let's have a look at file `examples/consuming-promise-data.ts` together

- Open the file, run it, watch how the logs appear

Composing Promises

So far we've looked at how to handle one async task, but how do we handle a sequence of async tasks each of which depend on the result of the one before?
For example, We make a request to a DB, then we make another request based on the result of the first request, and so on.
This process of chaining promises together is called composition. Promises are designed with composition in mind!

```
const fetchData = (result: number = 0): Promise<number> =>
  new Promise((resolve) => {
    setTimeout(() => resolve(result + 1), 1000)
  })

fetchData().then(fetchData).then(fetchData)
  .then(console.log) // 3
```

Emoji Check:

Do you understand how we can handle a sequence of async tasks using composition?

1. 😢 Haven't a clue, please help!
2. 🙁 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

Common Mistakes

We can achieve the same result as before by nesting our callbacks.
This is manageable to a point but we can quickly end up in a mess known affectionately by developers as "Callback Hell".

```
const fetchData = (result = 0): Promise<number> =>
  new Promise((resolve) => {
    setTimeout(() => resolve(result + 1), 1000)
  })

fetchData().then((result) => {
  fetchData(result).then((result) => {
    fetchData(result).then((result) => {
      console.log(result) // 3
    })
  })
})
```

Avoid this whenever possible.

Concurrent Promises - all

Let's say we have a number of async tasks to complete, but none of them directly depend on each other so we're not concerned with their order. All we need to know is that they have all completed, and we need their results.

```
const waitOne = (): Promise<number> =>
  new Promise((resolve) => {
    setTimeout(() => resolve(1), 1000)
  })

const waitTwo = (): Promise<number> =>
  new Promise((resolve) => {
    setTimeout(() => resolve(2), 2000)
  })

const tasks = [waitOne(), waitTwo(), waitOne()]

Promise.all(tasks).then(console.log) // [1,2,1]
```

Task - promise all

Promise.all(tasks) is a new promise, that resolves if every input promise resolves. It is rejected as soon as any of the input promises are rejected, and its output will be the first rejection message/error that occurs in the chain.

> Let's have a look at file `examples/concurrent-promises-with-all.ts` together

- Open the file, run it, watch how the logs appear

Emoji Check:

Do you understand how we can use `Promise.all()` to handle a number of promises that depend on each other?

1. 😢 Haven't a clue, please help!
2. 🙁 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

What about errors?

What if during the `all` there are error(s)?
As before, the chain will stop and the first rejection or error will be passed to any `catch` handler that we have defined.
...this means some data may be abandoned or lost! (This may be ok... it depends on your scenario)

Task - promise all with errors

Promise.all(tasks) is a new promise, that resolves if every input promise resolves. It is rejected as soon as any of the input promises are rejected, with the first rejection message/error

> Let's have a look at file `examples/concurrent-promises-with-errors.ts` together

- Open the file, run it, watch how the logs appear

Emoji Check:

Do you understand what happens with errors when we use `Promise.all()`?

1. 😢 Haven't a clue, please help!
2. 🙁 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

Concurrent Promises - allSettled

Rather than have the chain stop as soon as one promise has a problem, we can continue to process them all and have them all "settled", i.e. in a final state, using the `allSettled` method.

```
const waitOne = (): Promise<number> =>
  new Promise((resolve) => {
    setTimeout(() => resolve(1), 750)
  })

const boomBox = (): Promise<string> =>
  new Promise((resolve, reject) => {
    setTimeout(() => reject('Boom'), 500)
  })

const tasks = [waitOne(), boomBox(), waitOne()]

Promise.allSettled(tasks).then(console.log)
```

Output:

```
[
  { status: 'fulfilled', value: 1 },
  { status: 'rejected', reason: 'Boom' },
  { status: 'fulfilled', value: 1 }
]
```

Task - promise all settled

Promise.allSettled(tasks) is a new promise, that always resolves all the input promises.

> Let's have a look at file `examples/concurrent-promises-with-allSettled.ts` together

- Open the file, run it, watch how the logs appear
- Note how the output differs to `all` - gives us the outcomes of the the other functions as well as the resolved/rejected value!

Promise States

A Promise can be in one of three states: `pending | resolved | rejected`
We saw this in the previous task demo:

```
[
  { status: 'fulfilled', value: 1 },
  { status: 'fulfilled', value: 2 },
  { status: 'fulfilled', value: 1 },
  { status: 'rejected', reason: 'Database says no' }
]
```

It is rare in a project you will see any logs of the `pending` state.

Emoji Check:

Do you feel like you understand what `Promise.allSettled()` does, and how it differs from `Promise.all()`?

1. 😢 Haven't a clue, please help!
2. 🙁 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

Async / Await

ES6 introduced two new keywords for dealing with promises: `async` and `await`.
These try to mitigate "Callback Hell" by enabling us to write async code that looks synchronous.
There is nothing new here, rather using Async/Await is purely syntactic sugar, however it does make a huge difference to the readability of our code.

Here's some "Callback Hell":

```
const waitOne = (): Promise<number> =>
  new Promise((resolve) => {
    setTimeout(() => resolve(1), 1000)
  })

// Callback Hell
const waitThree = () =>
  new Promise((resolve) => {
    waitOne().then((a) => {
      waitOne().then((b) => {
        waitOne().then((c) => {
          resolve(a + b + c)
        })
      })
    })
  })

waitThree().then(console.log) // 3
```

Here's the `async/await` answer to that;

```typescript
const waitOne = (): Promise<number> =>
  new Promise((resolve) => {
    setTimeout(() => resolve(1), 1000)
  })

const waitThreeAsync = async () => {
  const a = await waitOne()
  const b = await waitOne()
  const c = await waitOne()
  return a + b + c
}

console.log(await waitThreeAsync()) // 3

// This still works
waitThreeAsync().then(console.log) // 3
```

What changed

We can replace any `Promise().then()` with `await Promise()` in the global scope. However, if we are awaiting a promise inside a function, we must decorate that function with `async`:

```
const asyncTask = async () => {
  return await someTask()
}

const result = await asyncTask()
```

Task - async/await

> Let's have a look at file `examples/async-await.ts` together

- Open the file, run it, watch how the logs appear

Error handling with try/catch

We have seen how to handle errors using `.catch`. TypeScript provides us with another method as well, known as `try/catch`.
Usually when we are using `async/await` syntax, we also use `try/catch` to handle errors.

```
try {
    // a function that potentially throws an error
    await someAsyncFunction() // Any time we need to wait
for a function to complete before moving on, we use the
'await' keyword
    console.log('Function call complete!')
} catch (error) {
    // this code handles exceptions
    console.log(error.message)
}
```

Try, catch and finally

If there is something we want our code to execute regardless of whether or not a function call returns an error, we can add a `finally` block to our `try/catch`.

```
try {
    // a function that potentially throws an error
    await someAsyncFunction()
} catch (error) {
    // this code handles exceptions
    console.log(error.message)
} finally {
  console.log('Done!')
}
```

Task - async/await

> Let's have a look at file `examples/try-catch.ts` together

- Open the file, run it, watch how the logs appear

Emoji Check:

How do you feel about the use of `async` and `await`?

1. 😢 Haven't a clue, please help!
2. 🙁 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

Overview - recap

- Error handling
- Composing Promises
- Common mistakes
- Concurrent Promises
- Async / Await
- Try / Catch
- Exercises

Objectives - recap

- Understand how to handle errors
- Understand how to compose promises
- Know some of the common mistakes
- Know how to use `async/await` keywords

Emoji Check:

On a high level, do you think you understand the main concepts of this session? Say so if not!

1. 😢 Haven't a clue, please help!
2. 🙁 I'm starting to get it but need to go over some of it please
3. 😐 Ok. With a bit of help and practice, yes
4. 🙂 Yes, with team collaboration could try it
5. 😀 Yes, enough to start working on it collaboratively

Speaker notes