Async Control Flow

By the end of this section, you should be able to:

- Understand native asynchronous primitives.
- Understand serial and parallel control flow with callbacks.
- Understand serial and parallel control flow with promises.
- Understand serial and parallel control flow with async/await.

Callbacks

A function that is called at some future point, once a task has been completed.

```
const {readFile} = require('fs')

readfile(__filename, (err, contents) ⇒ {
  if(err) {
    console.error(err)
    return
  }
  console.log(contents.toString())
})
```

Callbacks - Parallel Execution

A program with three variables, `smallFile`, `mediumFil` and `bigFile`.

```
1   const { readFile } = require('fs')
2   const [ bigFile, mediumFile, smallFile ] = Array.from(Array(3)).fill(__filename)
3
4   const print = (err, contents) ⇒ {
5     if (err) {
6       console.error(err)
7       return
8     }
9     console.log(contents.toString())
10  }
11  readFile(bigFile, print)
12  readFile(mediumFile, print)
13  readFile(smallFile, print)
```

Small file will be printed first, even though bigFile was called first.

This is a way to achieve parallel execution in Node.

Callbacks - Serial Execution

```
const { readFile } = require('fs')
const [ bigFile, mediumFile, smallFile ] = Array.from(Array(3)).fill(__filename)
const print = (err, contents) => {
  if (err) {
    console.error(err)
    return
  }
  console.log(contents.toString())
}
readFile(bigFile, (err, contents) => {
  print(err, contents)
  readFile(mediumFile, (err, contents) => {
    print(err, contents)
    readFile(smallFile, print)
  })
})
```

Serial execution is achieved by waiting for the callback before starting the next async operation.

What if we want all of the contents of each file to be concatenated?

```
const { readFile } = require('fs')
const [ bigFile, mediumFile, smallFile ] = Array.from(
const data = []
const print = (err, contents) => {
  if (err) {
    console.error(err)
    return
  }
  console.log(contents.toString())
}

readFile(bigFile, (err, contents) => {
  if (err) print(err)
  else data.push(contents)
  readFile(mediumFile, (err, contents) => {
    if (err) print(err)
    else data.push(contents)
    readFile(smallFile, (err, contents) => {
      if (err) print(err)
      else data.push(contents)
      print(null, Buffer.concat(data))
    })
  })
})
```

What about an unknown amount of async operations?

Using a self-recursive function with the two extra variables allows us to handle a list of any size.

```
const { readFile } = require('fs')
const files = Array.from(Array(3)).fill(__filename)
const data = []

const print = (err, contents) => {
  if (err) {
    console.error(err)
    return
  }
  console.log(contents.toString())
}

let count = files.length
let index = 0
const read = (file) => {
  readFile(file, (err, contents) => {
    index += 1
    if (err) print(err)
    else data.push(contents)
    if (index < count) read(files[index])
    else print(null, Buffer.concat(data))
  })
}

read(files[index])
```

fastseries

Callback-based serial execution can become quite complicated, quite quickly.

Using a small library to help with complexity is advised.

```
const { readFile } = require('fs')
const series = require('fastseries')()
const files = Array.from(Array(3)).fill(__filename)

const print = (err, data) => {
  if (err) {
    console.error(err)
    return
  }
  console.log(Buffer.concat(data).toString())
}

const readers = files.map((file) => {
  return (_, cb) => {
    readFile(file, (err, contents) => {
      if (err) cb(err)
      else cb(null, contents)
    })
  }
})

series(null, readers, null, print)
```

Promises

A promise represents an async operation that is either pending or settled.

If it's settled, it's either resolved or rejected.

With a callback:

```
function myAsyncOperation (cb) {
  doSomethingAsynchronous((err, value) ⇒ {
    cb(err, value)
  })
}

myAsyncOperation(functionThatHandlesTheResult)
```

With a Promise:

```
function myAsyncOperation () {
  return new Promise((resolve, reject) ⇒ {
    doSomethingAsynchronous((err, value) ⇒ {
      if (err) reject(err)
      else resolve(value)
    })
  })
}
```

The `promisify` function

```
const { promisify } = require('util')
const doSomething = promisify(doSomethingAsynchronous)

function myAsyncOperation () {
  return doSomething()
}

const promise = myAsyncOperation()
                  .then(value ⇒ console.log(value))
                  .catch(err ⇒ console.log(err))
```

A more concrete example

```
const { promisify } = require('util')
const { readFile } = require('fs')

const readFileProm = promisify(readFile)

const promise = readFileProm(__filename)

promise.then((contents) => {
  console.log(contents.toString())
})

promise.catch((err) => {
  console.error(err)
})
```

```
const { readFile } = require('fs').promises

readFile(__filename)
  .then((contents) => {
    console.log(contents.toString())
  })
  .catch(console.error)
```

Series operation

```
const { readFile } = require('fs').promises
const [ bigFile, mediumFile, smallFile ] = Array.from(Array(3)).fill(__filename)

const print = (contents) => {
  console.log(contents.toString())
}
readFile(bigFile)
  .then((contents) => {
    print(contents)
    return readFile(mediumFile)
  })
  .then((contents) => {
    print(contents)
    return readFile(smallFile)
  })
  .then(print)
  .catch(console.error)
```

Unknown number of files

```
const { readFile } = require('fs').promises
const files = Array.from(Array(3)).fill(__filename)
const data = []
const print = (contents) => {
  console.log(contents.toString())
}
let count = files.length
let index = 0
const read = (file) => {
  return readFile(file).then((contents) => {
    index += 1
    data.push(contents)
    if (index < count) return read(files[index])
    return data
  })
}

read(files[index])
  .then((data) => {
    print(Buffer.concat(data))
  })
  .catch(console.error)
```

Promise.all()

```
const { readFile } = require('fs').promises
const files = Array.from(Array(3)).fill(__filename)
const print = (data) => {
  console.log(Buffer.concat(data).toString())
}

const readers = files.map((file) => readFile(file))

Promise.all(readers)
  .then(print)
  .catch(console.error)
```

Slight problem here is that if one of the Promises fails, it all fails.

Promise.allSettled()

```
const { readFile } = require('fs').promises
const files = [filename, 'not a file', filename]

const print = (results) ⇒ {
  results
    .filter(({status}) ⇒ status = 'rejected')
    .forEach(({reason}) ⇒ console.error(reason))
  const data = results
    .filter(({status}) ⇒ status = 'fulfilled')
    .map(({value}) ⇒ value)
  const contents = Buffer.concat(data)
  console.log(contents.toString())
}

const readers = files.map((file) ⇒ readFile(file))

Promise.allSettled(readers)
  .then(print)
  .catch(console.error)
```

Promises in Parallel

Either use `allSettled` or give each their own then/catch handlers.

```
const { readFile } = require('fs').promises
const [ bigFile, mediumFile, smallFile ] = Array.from(Array(3)).fill(__filename)

const print = (contents) => {
  console.log(contents.toString())
}

readFile(bigFile).then(print).catch(console.error)
readFile(mediumFile).then(print).catch(console.error)
readFile(smallFile).then(print).catch(console.error)
```

Async/Await

- Stylistically similar to sync code.

```
const { readFile } = require('fs').promises

async function run () {
  const contents = await readFile(__filename)
  console.log(contents.toString())
}

run().catch(console.error)
```

Series in async/await

```
const { readFile } = require('fs').promises

const print = (contents) ⇒ {
  console.log(contents.toString())
}
const [ bigFile, mediumFile, smallFile ] = Array.from(Array(3)).fill(__filename)

async function run () {
  print(await readFile(bigFile))
  print(await readFile(mediumFile))
  print(await readFile(smallFile))
}

run().catch(console.error)
```

Concatenate

```
const { readFile } = require('fs').promises
const print = (contents) ⇒ {
  console.log(contents.toString())
}
const [ bigFile, mediumFile, smallFile ] = Array.from(Array(3)).fill(__filename)

async function run () {
  const data = [
    await readFile(bigFile),
    await readFile(mediumFile),
    await readFile(smallFile)
  ]
  print(Buffer.concat(data))
}

run().catch(console.error)
```

Unknown length?

```
const { readFile } = require('fs').promises

const print = (contents) ⇒ {
  console.log(contents.toString())
}

const files = Array.from(Array(3)).fill(__filename)

async function run () {
  const data = []
  for (const file of files) {
    data.push(await readFile(file))
  }
  print(Buffer.concat(data))
}

run().catch(console.error)
```

This is the right approach where the operations must be sequentially called.

Output order matters, Execution order doesn’t

```
const { readFile } = require('fs').promises
const files = Array.from(Array(3)).fill(__filename)
const print = (contents) ⇒ {
  console.log(contents.toString())
}

async function run () {
  const readers = files.map((file) ⇒ readFile(file))
  const data = await Promise.all(readers)
  print(Buffer.concat(data))
}

run().catch(console.error)
```

Parallel execution with sequentially ordered output.

Same problem with the Promise.all() as before

Use allSettled()

```
const { readFile } = require('fs').promises
const files = [filename, 'foo', filename]
const print = (contents) => {
  console.log(contents.toString())
}

async function run () {
  const readers = files.map((file) => readFile(file))
  const results = await Promise.allSettled(readers)

  results
    .filter(({status}) => status === 'rejected')
    .forEach(({reason}) => console.error(reason))

  const data = results
    .filter(({status}) => status === 'fulfilled')
    .map(({value}) => value)

  print(Buffer.concat(data))
}

run().catch(console.error)
```

Exercises

1. In the labs folder, there is a file `parallel.js`. The functions must be called in the order `opA`, `opB` and `opC`.

Call them in such a way so that `C` then `B` then `A` is printed out.

2. In the labs folder, there is a file `serial.js`. Call the functions in such a way such that `A` then `B` then `C` is printed out.

3. In `lab.js` use the `api.fetch()` function to complete the two exercises. How many different ways can you do it in? Explore some parallel and series approaches.