

1 The Node Binary

1.1 Common Command Line Arguments

```
1  # only check syntax
2  node --check app.js
3  node -c app.js
4
5  # evaluate (but don't print)
6  node --eval "1+1"
7  node -e "console.log(1+1)"
8  2
9  node -e "console.log(1+1); 0"
10 2
11
12 # evaluate and print
13 node -e "console.log(1+1)"
14 2
15 undefined
16
17 node -p "console.log(1+1); 0"
18 2
19 0
```

1.2 Module availability

All Node core modules can be accessed by their namespaces within the code evaluation context - no require required:

```
1  node -p "fs.readdirSync('.').filter((f) => /\.js$/).test(f)"
2  []
```

1.3 Preloading files

```
1  // preload.js
2  console.log('preload.js: this is preloaded')
3
4  // app.js
5  console.log('app.js: this is the main file')
6
```

```
1  // CommonJS
2  node -r ./preload.js app.js
3  node --require ./preload.js app.js
4
```

```
5 // ES modules
6 node --loader ./preload.js app.js
```

1.4 Stack trace limit

By default, only the first 10 stack frames are shown, which can lead to the root cause of the error not being shown.

In this case, modify the V8 option `--stack-trace-limit`:

```
1 node --stack-trace-limit=20 file.js
```

2 Debugging and Diagnostics

Start node in debugging mode:

```
1 node --inspect file.js # runs immediately
2 node --inspect-brk file.js # breakpoint at start of program
```

3 Core JavaScript Concepts

3.1 Types

Everything besides the following primitive types is an object - functions and arrays, too, are objects.

3.1.1 Primitive Types

```
1 // The null primitive is typically used to describe
2 // the absence of an object...
3 // Null
4 null
5
6 // Undefined
7 // ... whereas undefined is the absence
8 // of a defined value.
9 // Any variable initialized without a value will be undefined.
10 // Any expression that attempts access of a non-existent
11 // property on an object will result in undefined.
12 // A function without a return statement will return undefined.
13 // undefined
14
15 // Number
16 // The Number type is double-precision floating-point format.
17 // It allows both integers and decimals but
```

```

18 // has an integer range of  $-2^{53}-1$  to  $2^{53}-1$ .
19 1, 1.5, -1e4, NaN
20
21 // BigInt
22 // The BigInt type has no upper/lower limit on integers.
23 1n, 9007199254740993n
24
25 // String
26 'str', "str", `str ${var}`
27
28 // Boolean
29 true, false
30
31 // Symbol
32 // Symbols can be used as unique identifier keys in objects.
33 //The Symbol.for method creates/gets a global symbol.
34 Symbol('description'), Symbol.for('namespace')...
35
36

```

3.1.2 Object

An object is a set of key value pairs, where values can be any primitive type or an object (including functions, since functions are objects). Object keys are called properties.

All JavaScript objects have prototypes. A prototype is an implicit reference to another object that is queried in property lookups. If an object doesn't have a particular property, the object's prototype is checked for that property. and so on. This is how inheritance in JavaScript works.

3.1.3 Functions

```

1 // this refers to the object on which the function was called,
2 // not the object the function was assigned to
3 const obj = { id: 999, fn: function () { console.log(this.id) } }
4 const obj2 = { id: 2, fn: obj.fn }
5 obj2.fn() // prints 2
6 obj.fn() // prints 999
7
8 // Functions have a call method that can be used
9 // to set their this context. See
10 /**
11 * Calls a method of an object,

```

```

12      * substituting another object for the current object.
13      * @param thisArg The object to be used as the current object.
14      * @param argArray A list of arguments to be passed to the method.
15      */
16      call(this: Function, thisArg: any, ...argArray: any[]): any;
17
18      function fn() { console.log(this.id) }
19      const obj = { id: 999 }
20      const obj2 = { id: 2 }
21      fn.call(obj2) // prints 2
22      fn.call(obj) // prints 999
23
24      /*
25      *Lambda functions do not have their own this context.
26      * Wwhen this is *referenced inside a function,
27      * it refers to the this of the
28      *nearest parent non-lambda function.
29      */
30      function fn() {
31          return (offset) => {
32              console.log(this.id + offset)
33          }
34      }
35      const obj = { id: 999 }
36      const offsetter = fn.call(thisArg = obj)
37      console.log(typeof(offsetter)); // function
38      offsetter(1) // 1000
39

```

Lambda functions do not have a prototype.

3.2 Prototypal Inheritance

3.2.1 Prototypal inheritance - functional

Create a prototype chain:

```

1      const wolf = {
2          howl: function () { console.log(this.name + ': awoooooooooo') }
3      }
4
5      const dog = Object.create(wolf, {
6          woof: { value: function() { console.log(this.name + ': woof') } }
7      })

```

```

8
9     const rufus = Object.create(dog, {
10         name: {value: 'Rufus the dog'}
11     })
12
13     rufus.woof() // prints "Rufus the dog: woof"
14     rufus.howl() // prints "Rufus the dog: awoooooooooo"

```

A Property Descriptor is a JavaScript object that describes the characteristics of the properties on another object.

```

1     const propdesc = Object.getOwnPropertyDescriptors(rufus);
2     console.log(propdesc);
3
4     // output
5     {
6         name: {
7             value: 'Rufus the dog',
8             writable: false,
9             enumerable: false,
10            configurable: false
11        }
12    }
13
14
15     const name = Object.getOwnPropertyDescriptor(rufus, "name");
16     console.log(name);
17
18     // output
19
20     {
21         value: 'Rufus the dog',
22         writable: false,
23         enumerable: false,
24         configurable: false
25     }
26

```

3.2.2 Prototypal inheritance - using constructor

Creating an object with a specific prototype object can also be achieved by calling a function with the new keyword.

The constructor approach to creating a prototype chain is to define properties on a function's prototype object and then call that function with new.

Define how to create the parent object:

```
1  function Wolf (name) {  
2      this.name = name  
3  }  
4  
5  Wolf.prototype.howl = function () {  
6      console.log(this.name + ': awoooooooooo')  
7  }
```

Define a function to set up the inheritance chain:

```
1  function inherit (proto) {  
2      function ChainLink(){}  
3      ChainLink.prototype = proto  
4      return new ChainLink()  
5  }
```

Define how to obtain a child object:

```
1  function Dog (name) {  
2      Wolf.call(this, name + ' the dog')  
3  }  
4  
5  Dog.prototype = inherit(Wolf.prototype)  
6  
7  Dog.prototype woof = function () {  
8      console.log(this.name + ': woof')  
9  }  
10
```

Create a child object():

```
1  const rufus = new Dog('Rufus')  
2  
3  rufus.woof() // prints "Rufus the dog: woof"  
4  rufus.howl() // prints "Rufus the dog: awoooooooooo"
```

In JavaScript runtimes that support EcmaScript 5+ the Object.create function could be used to the same effect:

```

1
2  function Dog (name) {
3      Wolf.call(this, name + ' the dog')
4  }
5
6  Dog.prototype = Object.create(Wolf.prototype)
7
8  Dog.prototype woof = function () {
9      console.log(this.name + ': woof')
10 }

```

Node.js has a utility function: `util.inherits` that is often used in code bases using constructor functions.

```

1
2  const util = require('util')
3
4  function Dog (name) {
5      Wolf.call(this, name + ' the dog')
6  }
7
8  Dog.prototype.woof = function () {
9      console.log(this.name + ': woof')
10 }
11
12 // sets the prototype of Dog.prototype to Wolf.prototype
13 util.inherits(Dog, Wolf)
14

```

In contemporary Node.js, `util.inherits` uses the EcmaScript 2015 (ES6) method `Object.setPrototypeOf` under the hood.

```

1  Object.setPrototypeOf(Dog.prototype, Wolf.prototype)

```

3.2.3 Prototypal inheritance - class-based

The `class` keyword is syntactic sugar that actually creates a function, `new()`, that is to be used as a constructor. Internally, it creates prototype chains.

Usage:

```

1  class Wolf {
2      constructor (name) {
3          this.name = name
4      }
5      howl () { console.log(this.name + ': awoooooooooo') }
6  }
7
8  class Dog extends Wolf {
9      constructor(name) {
10         super(name + ' the dog')
11     }
12     woof () { console.log(this.name + ': woof') }
13 }
14
15 const rufus = new Dog('Rufus')

```

3.3 Inheritance provided by closures

Below, the spread operator is used. The spread operator splits arrays into their members, and object into properties. When spreading objects, the properties are added as key-value pairs.

```

1  function wolf (name) {
2      const howl = () => {
3          console.log(name + ': awoooooooooo')
4      }
5      return { howl: howl }
6  }
7
8  function dog (name) {
9      name = name + ' the dog'
10     const woof = () => { console.log(name + ': woof') }
11     return {
12         ...wolf(name),
13         woof: woof
14     }
15 }
16 const rufus = dog('Rufus')
17
18 console.log(rufus)
19 // { howl: [Function: howl], woof: [Function: woof] }
20 rufus.woof()

```



```

21 // "Rufus the dog: woof"
22 rufus.howl()
23 // "Rufus the dog: awoooooooooooo"

```

4 Packages and Dependencies

4.1 Specifying a SemVer range

- Prefix the version with a caret (^) to include everything that does not increment the first non-zero portion of semver. Example: ^8.14.1 is the same as 8.x.x.
Note: caret behavior is different for 0.x versions, for which it will only match patch versions.
- Use the tilde symbol to include everything greater than a particular version in the same minor range. Example: ~2.2.0 matches 2.2.0 and 2.2.1 (highest existing minor version).
- Specify a range of versions. Example: >2.1 matches 2.2.0, 2.2.1, ... Note: There must be spaces on either side of hyphens.
- Use || to combine multiple sets of versions. Example: ^2 <2.2 || > 2.3.

5 Module System

5.1 Detecting Main Module in CJS

In some situations we may want a module to be able to operate both as a program and as a module that can be loaded into other modules.

When a file is the entry point of a program, it's the main module. We can detect whether a particular file is the main module.

```

1 // imports
2
3 if (require.main === module) {
4   // ...
5 } else {
6   const myfunc = (str) => {
7     // ...
8   }
9   module.exports = myfunc
10 }

```

The "start" script in the package.json file executes node index.js. When a file is called with node that file is the *entry point of a program*.

```
1   npm start
2
3   // or:
4   node index.js
5
6   // app starts
```

If it is loaded as a module, it will export function myfunc:

```
1   $node -p 'require("./index.js")'
2   [Function: myfunc]
3
4   $ node -p 'require("./index.js")("test")'
5   TSET
```

5.2 Converting a Local CJS File to a Local ESM File

When using ECMA Script modules in a CJS application, module files need to have the .mjs extension.

```
1   node -e "fs.renameSync('./format.js', './format.mjs');"
2   node -p "fs.readdirSync('.').join('\t');"

require() cannot be used with ESM modules.
```

Instead, we need to change it to a dynamic import() which is available in all CommonJS modules.

This is due to CJS loading modules synchronously, while ESM loads modules asynchronously. As a consequence, ESM can import CJS, but CJS cannot require ESM since that would break the synchronous constraint.

5.2.1 Aside: Static and Dynamic Imports

Assume we have a file utils.mjs

```
1   // Default export
2   export default () => {
3       console.log('Hi from the default export!');
4   };
5
6   // Named export `doStuff`
7   export const doStuff = () => {
8       console.log('Doing stuff');
9   };
```

This is a static import:

```
1 import * as module from './utils.mjs';
```

Whereas this is a dynamic import:

```
1 // returns a Promise
2 import('./utils.mjs')
3 .then((module) => {
4     module.default();
5     // logs 'Hi from the default export!'
6     module.doStuff();
7     // logs 'Doing stuff'
8 });
9
10 // or
11 (async () => {
12     const moduleSpecifier = './utils.mjs';
13     const module = await import(moduleSpecifier)
14     module.default();
15     // logs 'Hi from the default export!'
16     module.doStuff();
17     // logs 'Doing stuff'
18 })();
19
```

5.3 Converting a CJS Package to an ESM Package

5.3.1 Specify module type

We can opt-in to ESM-by-default by adding a `type` field to the `package.json` and setting it to `"module"`. Our `package.json` should look as follows:

```
1 {
2     "name": "my-package",
3     "version": "1.0.0",
4     "main": "index.js",
5     "type": "module",
6     //...
7 }
8
```

5.3.2 Exports in ESM

Whereas in CJS, we assigned a function to `module.exports`:

```
1 module.exports = myfunc
```

in ESM we use the `export default` keyword and follow with a function expression to set a function as the main export:

```
1 export default (str) => {  
2   return somefunc(str);  
3 }
```

The default exported function is synchronous again, as it should be.

Note: ESM exports must be statically analyzable; and this means they can't be conditionally declared. The `export` keyword only works at the top level.

EcmaScript Modules were primarily specified for browsers, implying that there is no concept of a main module in the spec (since modules are initially loaded via HTML, which could allow for multiple script tags).

We can however infer that a module is the first module executed by Node by comparing `process.argv[1]` (which contains the execution path of the entry file) with `import.meta.url`.

```
1 const isMain = process.argv[1] &&  
2   await realpath(fileURLToPath(import.meta.url)) ===  
3   await realpath(process.argv[1])
```

One compelling feature of modern ESM is *Top-Level Await (TLA)*. Since all ESM modules load asynchronously it's possible to perform related asynchronous operations as part of a module's initialization.

TLA allows the use of the `await` keyword in an ESM modules scope at the top level, in addition to the standard usage within `async` functions.

With a dynamic import, if we want to use an imported module as default export, we have to reassign the `default` property to it. That's because dynamic imports return a promise which resolves to an object. If there's a default export in a module, the `default` property of that object will be set to it.

```
1 if (isMain) {  
2   const { default: pino } = await import('pino')  
3   const logger = pino()  
4   //  
5 }  
6  
7 export default (str) => {
```

```

8     return format.upper(str).split('').reverse().join('')
9 }

```

5.3.3 Importing modules

With static imports, different import possibilities exist:

- Implicitly import a module's default export

```

1     // the default export of the url module is assigned
2     // to the url reference.
3     import url from 'url'
4

```

- Import a specific named export from a module

```

1     import { realpath } from 'fs/promises'
2

```

- If there are no default exports, just individual ones, the following syntax is used:

```

1     import * as format from './format.js'
2

```

If a module *does* have a default export and that same syntax - `import * as` - is used to load it, the resulting object will have a default property holding the default export.

Note: ESM does not support loading modules without a full file extension.

5.4 Resolving a Module Path in CJS

The `require` function has a method called `require.resolve`. This can be used to determine the absolute path for any required module.

Example:

```

1     # package resolution
2     # no path given: looks into node-modules
3     require('pino') => /home/key/code/[...]/app/node_modules/pino/pino.js
4     require('standard') => /[...]/app/node_modules/standard/index.js

```

```

5
6   # directory resolution
7   # resolves to index.js!
8   require('.') => /home/key/code/[...]/app/index.js
9   require('../app') => /home/key/code/[...]/app/index.js
10
11  # file resolution
12  # path given: resolves to local file
13  # both with and without extension work
14  require('./format') => /home/key/code/[...]/app/format.js
15  require('./format.js') => /home/key/code/[...]/app/format.js
16
17  # core APIs resolution
18  require('fs')      => fs
19  require('util')    => util
20

```

5.5 Resolving a Module Path in ESM

Currently there is experimental support for an `import.meta.resolve` function which returns a promise that resolves to the relevant file:// URL for a given valid input. Since this is experimental, and behind the `--experimental-import-meta-resolve` flag, we'll discuss an alternative approach to module resolution inside an EcmaScript Module.

We can use the ecosystem `import-meta-resolve` module to get the best results for now.

```

import resolve from 'import-meta-resolve'
console.log( 'import pino', '=', await resolve('pino', import.meta.url) )
console.log( 'import tap', '=', await resolve('tap', import.meta.url) )

```

```

1   import { resolve } from 'import-meta-resolve'
2
3   console.log(
4     `import 'pino`,
5     '=>',
6     await resolve('pino', import.meta.url)
7   )
8
9   // If a package's package.json exports field defines
10  // an ESM entry point, the require.resolve function will still
11  // resolve to the CJS entry point because require is a CJS API.
12  // import-meta-resolve has a workaround
13  console.log(

```

```

14   `import 'tap'`,
15   '=>',
16   await resolve('tap', import.meta.url)
17   )
18
19   // resolved to [...]tap/dist/esm/index.js
20
21

```

6 Asynchronous Control Flow

6.1 Callbacks

Here, the `readFile` function schedules a task, which is to read the given file. When the file has been read, the `readFile` function will call the function provided as the second argument.

The second argument to `readFile` is a function that has two parameters, `err` and `contents`. This function will be called when `readFile` has completed its task. If there was an error, then the first argument passed to the function will be an error object representing that error, otherwise it will be null. If the `readFile` function is successful, the first argument (`err`) will be null and the second argument (`contents`) will be the contents of the file.

```

1   const { readFile } = require('fs')
2
3   // __filename holds the path of the file currently being executed
4   readFile(__filename, (err, contents) => {
5       if (err) {
6           console.error(err)
7           return
8       }
9       console.log(contents.toString())
10  })

```

This yields a way to achieve parallel execution in Node.js:

```

1   const { readFile } = require('fs')
2   const [ bigFile, mediumFile, smallFile ] =
3       Array.from(Array(3)).fill(__filename)
4
5   const print = (err, contents) => {
6       if (err) {

```

```

7         console.error(err)
8         return
9     }
10    console.log(contents.toString())
11  }
12  readFile(bigFile, print)
13  readFile(mediumFile, print)
14  readFile(smallFile, print)

```

Here the smallest file will be printed first, even though it's scheduled to be read last.

If instead we wanted to use serial execution, let's say we wanted bigFile to print first, then mediumFile even though they take longer to load than smallFile, we'd have to place the callbacks inside each other:

```

1  readFile(bigFile, (err, contents) => {
2    print(err, contents)
3    readFile(mediumFile, (err, contents) => {
4      print(err, contents)
5      readFile(smallFile, print)
6    })
7  })

```

Thus, serial execution with callbacks is achieved by waiting for the callback to call before starting the next asynchronous operation.

6.2 Promises

A promise is an object that represents an asynchronous operation. It's either pending or settled, and if it is settled it's either resolved or rejected.

Being able to treat an asynchronous operation as an object is a useful abstraction. For instance, instead of passing a function that should be called when an asynchronous operation completes into another function (e.g., a *callback*), a *promise* that represents the asynchronous operation can be returned from a function instead.

This is a callback-based approach:

```

1
2  function myAsyncOperation (cb) {
3    doSomethingAsynchronous((err, value) => { cb(err, value) })
4  }
5
6  myAsyncOperation(functionThatHandlesTheResult)
7

```

This is the same in promise form:

```
1
2  function myAsyncOperation () {
3      return new Promise((resolve, reject) => {
4          // doSomethingAsynchronous expects a callback
5          doSomethingAsynchronous((err, value) => {
6              if (err) reject(err)
7              else resolve(value)
8          })
9      })
10 }
11
12 const promise = myAsyncOperation()
13 // next up: do something with promise
14
15
```

This gets a lot nicer with the `promisify` function from the `util` module:

```
1
2  const { promisify } = require('util')
3  const doSomething = promisify(doSomethingAsynchronous)
4  function myAsyncOperation () {
5      return doSomething()
6  }
7
8  const promise = myAsyncOperation()
9
```

Promise success or failure are handled using `then` and `catch`:

```
1
2  const promise = myAsyncOperation()
3  // then and catch always return a promise, so they can be chained
4  promise
5  .then((value) => { console.log(value) })
6  .catch((err) => { console.error(err) })
7
```

Below, we have the same `readFile` operation as in the last section, but the `promisify` function is used to convert a callback-based API to a promise-based one.

```
1
2  const { promisify } = require('util')
3  const { readFile } = require('fs')
4
5  const readFileProm = promisify(readFile)
6
7  const promise = readFileProm(__filename)
8
9  promise.then((contents) => {
10     console.log(contents.toString())
11 })
12
13 promise.catch((err) => {
14     console.error(err)
15 })
```

However, using `promisify` with `fs` is not necessary, since `fs` already exports a promises object with promise-based versions. Using this, we can write

```
1
2  const { readFile } = require('fs').promises
3
4  readFile(__filename)
5    .then((contents) => {
6      console.log(contents.toString())
7    })
8    .catch(console.error)
```

Here, even though an intermediate promise is created by the first `then`, we still only need the one `catch` handler, as rejections are propagated.

Promises also allow for an easy serial execution pattern:

```
1
2  readFile(bigFile)
3  // returns a promise for reading mediumFile
4  .then((contents) => {
5    print(contents)
6    return readFile(mediumFile)
7  })
8  //returns a promise for reading smallFile
```

```

9     .then((contents) => {
10         print(contents)
11         return readFile(smallFile)
12     })
13     // returns itself
14     .then(print)
15     .catch(console.error)
16

```

If parallel execution is desired, `Promise.all` can be used to wait for all tasks to be handled. `Promise.all` takes an array of promises and returns a promise that resolves when all promises have been resolved. That returned promise resolves to an array of the values for each of the promises. This will give the same result of asynchronously reading all the files and concatenating them in a prescribed order.

```

1
2     const readers = files.map((file) => readFile(file))
3
4     Promise.all(readers)
5         .then(print)
6         .catch(console.error)
7

```

If one of the promises were to fail, `Promise.all` would reject, and any successfully resolved promises are ignored. If we want more tolerance of individual errors, `Promise.allSettled` can be used:

```

1
2     const { readFile } = require('fs').promises
3     const files = [__filename, 'not a file', __filename]
4     const print = (results) => {
5         results
6             .filter(({status}) => status === 'rejected')
7             .forEach(({reason}) => console.error(reason))
8         const data = results
9             .filter(({status}) => status === 'fulfilled')
10            .map(({value}) => value)
11         const contents = Buffer.concat(data)
12         console.log(contents.toString())
13     }
14
15     const readers = files.map((file) => readFile(file))
16

```

```
17 Promise.allSettled(readers)
18   .then(print)
19   .catch(console.error)
20
```

The `Promise.allSettled` function returns an array of objects representing the settled status of each promise. Each object has a `status` property, which may be `rejected` or `fulfilled`. Objects with a `rejected` status will contain a `reason` property containing the error associated with the rejection. Objects with a `fulfilled` status will have a `value` property containing the resolved value.

Finally, if we want promises to run in parallel independently, we can either use `Promise.allSettled` or simply execute each of them with their own `then` and `catch` handlers:

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

6.3 Async/Await

An `async` function always returns a promise. The promise will resolve to whatever is returned inside the `async` function body.

The `await` keyword can only be used inside of `async` functions. Calling `await` will pause the execution of the `async` function until the awaited promise is resolved. The resolved value of that promise will be returned from an `await` expression.

Here's an example of the same `readFile` operation from the previous section, but this time using an `async` function:

```
1
2 const { readFile } = require('fs').promises
3
4 async function run () {
5   const contents = await readFile(__filename)
6   console.log(contents.toString())
7 }
8
9 run().catch(console.error)
10
```

An `async` function always returns a promise, so we call the `catch` method to ensure that any rejections within the `async` function are handled.

This is how serial execution would work with async/await. Both variants print ABC:

```
1
2  async function run () {
3      print(await readFile(bigFile))
4      print(await readFile(mediumFile))
5      print(await readFile(smallFile))
6  }
7
8  run().catch(console.error)
9
```

Here is a serial execution example promisifying a "normal" function:

```
1 // see https://yieldcode.blog/post/implementing-promisable-set-timeout/
2
3 function setTimeoutPromise(cb, ms) {
4     return new Promise((resolve) => {
5         setTimeout(() => resolve(cb()), ms);
6     });
7 }
8
9 const print = (err, contents) => {
10     if (err) console.error(err)
11     else console.log(contents )
12 }
13
14 const opA = async (cb) => {
15     //setTimeout(() => {
16         await setTimeoutPromise(() => {
17             cb(null, 'A')
18         }, 500)
19     }
20 }
21
22 const opB = async (cb) => {
23     await setTimeoutPromise(() => {
24         cb(null, 'B')
25     }, 250)
26 }
27
28 const opC = async (cb) => {
29     await setTimeoutPromise(() => {
30         cb(null, 'C')
```

```

31     }
32
33 function sleep(delay) {
34     return new Promise(resolve => setTimeout(resolve, delay));
35 }
36
37 const run = async () => {
38     // option 1, using then()
39     opA(print).then(() => {
40         // opC needs to be called inside this!!
41         opB(print).then(() => {
42             opC(print)})
43     })
44 };
45
46 await sleep(3000);
47
48 // option 2, using await
49 await opA(print);
50 await opB(print);
51 await opC(print);
52 }
53
54 run()

```

If the output only has to be ordered, but the order in which asynchronous operations resolves is immaterial, we can again use `Promise.all` but this time await the promise that `Promise.all` returns:

```

1
2 async function run () {
3     const readers = files.map((file) => readFile(file))
4     const data = await Promise.all(readers)
5     print(Buffer.concat(data))
6 }
7
8 run().catch(console.error)
9

```

To get the exact same parallel operation behavior as in the initial callback example, so that the files are printed as soon as they are loaded, we have to create the promises, use a `then` handler and then await the promises later on:

```

1
2  async function run () {
3      const big = readFile(bigFile)
4      const medium = readFile(mediumFile)
5      const small = readFile(smallFile)
6
7      big.then(print)
8      medium.then(print)
9      small.then(print)
10
11     await small
12     await medium
13     await big
14 }
15
16 run().catch(console.error)
17

```

This will ensure the contents are printed out chronologically, according to the time it took each of them to load.

To get that behavior with the timeout example, we rewrite the code as follows. Note no then handlers are needed:

```

1  // see https://yieldcode.blog/post/implementing-promisable-set-timeout/
2
3  function setTimeoutPromise(cb, ms) {
4      return new Promise((resolve) => {
5          setTimeout(() => resolve(cb()), ms);
6      });
7  }
8
9  const print = (err, contents) => {
10     if (err) console.error(err)
11     else console.log(contents )
12 }
13
14 const opA = async (cb) => {
15     //setTimeout(() => {
16         await setTimeoutPromise(() => {
17             cb(null, 'A')
18         }, 500)
19     }
20

```

```

21     const opB = async (cb) => {
22         await setTimeoutPromise(() => {
23             cb(null, 'B')
24         }, 250)
25     }
26
27     const opC = async (cb) => {
28         await setTimeoutPromise(() => {
29             cb(null, 'C')
30         }, 125)
31     }
32
33     const run = async () => {
34
35         const A = opA(print);
36         const B = opB(print);
37         const C = opC(print);
38
39         await A;
40         await B;
41         await C;
42
43     }
44
45     run()
46

```

If the complexity for parallel execution grows it may be better to use a callback based approach and wrap it at a higher level into a promise so that it can be used in an async/await function:

```

1
2     const { promisify } = require('util')
3     const { readFile } = require('fs')
4     const [ bigFile, mediumFile, smallFile ] =
5         Array.from(Array(3)).fill(__filename)
6
7     const read = promisify((cb) => {
8         let index = 0
9         const print = (err, contents) => {
10             index += 1
11             if (err) {
12                 console.error(err)

```



```

13         if (index === 3) cb()
14         return
15     }
16     console.log(contents.toString())
17     if (index === 3) cb()
18 }
19 readFile(bigFile, print)
20 readFile(mediumFile, print)
21 readFile(smallFile, print)
22 })
23
24 async function run () {
25     await read()
26     console.log('finished!')
27 }
28
29 run().catch(console.error)

```

Here the read function returns a promise that resolves when all three parallel operations are done.

6.4 Canceling Asynchronous Operations

To cancel asynchronous operations, Node core has embraced the [AbortController](#) with AbortSignal Web APIs.

While AbortController with AbortSignal can be used for callback-based APIs, it's generally used in Node to solve for the fact that promise-based APIs return promises:

```

1
2     import { setTimeout } from 'timers/promises'
3
4     const ac = new AbortController()
5     const { signal } = ac
6     const timeout = setTimeout(1000, 'will NOT be logged', { signal })
7
8     setImmediate(() => {
9         ac.abort()
10    })
11
12    try {
13        console.log(await timeout)
14    } catch (err) {
15        // ignore abort errors:

```

```
16     if (err.code !== 'ABORT_ERR') throw err
17   }
18
```

The `AbortController` constructor is a global, so we instantiate it and assign it to the `ac` constant. An `AbortController` instance has an `AbortSignal` instance on its `signal` property. We pass this via the `options` argument to timers/promises `setTimeout`; internally the API will listen for an abort event on the signal instance and then cancel the operation if it is triggered.

Many parts of the Node core API accept a `signal` option, including `fs`, `net`, `http`, `events`, `child_process`, `readline` and `stream`.

7 Appendix

7.1 JavaScript Function Definitions

From: <https://dmitripavlutin.com/6-ways-to-declare-javascript-functions/>

7.1.1 Function declaration

A function *declaration* starts with `function`.

An important property of this syntax is its *hoisting* mechanism. Hoisting allows the function to be used before its declaration.

```
1  function isNil(value) {
2    return value == null;
3  }
```

7.1.2 Function expression (unnamed)

An unnamed *expression* containing the word *function*.

```
1
2  // create a method on an object
3  { sum: function() {...},
4    //
5  }
6
7  // use as a callback
8  array.map(function(...) {...})
9
```

7.1.3 Function expression (named)

A named *expression* containing the word *function*.

```
1
2  const isTruthy = function(value) {
3      return !!value;
4  }
5  // or
6  let count = function(...) {...}
7
```

7.1.4 Shorthand method definition

Like a function definition/expression, but missing the word *function*.

```
1
2  const collection = {
3      items: [],
4      add(...items) {
5          this.items.push(...items);
6      },
7      get(index) {
8          return this.items[index];
9      }
10 };
11
12 collection.add('C', 'Java', 'PHP');
13
```

7.1.5 Arrow function

The arrow function does not itself create its execution context, but takes it lexically (contrary to function expression or function declaration, which create their own this depending on invocation).

```
1
2  const absValue = (number) => {
3      if (number < 0) {
4          return -number;
5      }
6      return number;
7  }
```

7
8

```
}
```

7.1.6 Generator function

The generator function in JavaScript returns a Generator object. Its syntax is similar to function expression, function declaration, or method declaration, but it requires a star character *.

1
2
3
4
5
6
7
8
9
10

```
const obj = {  
  *indexGenerator() {  
    var index = 0;  
    while(true) {  
      yield index++;  
    }  
  }  
}
```

1

1

1

1