1 The Node Binary

1.1 Common Command Line Arguments

```
# only check syntax
node --check app.js
node -c app.js

# evaluate (but don't print)
node --eval "1+1"
node -e "console.log(1+1)"
2
node -e "console.log(1+1); 0"
2

# evaluate and print
node -e "console.log(1+1)"
2
undefined

node -p "console.log(1+1); 0"
2
0
```

1.2 Module availability

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

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

1.3 Preloading files

```
// preload.js
console.log('preload.js: this is preloaded')

// app.js
console.log('app.js: this is the main file')

```

```
// CommonJS
node -r ./preload.js app.js
node --require ./preload.js app.js

```

```
    // ES modules
    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:

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

2 Debugging and Diagnostics

Start node in debugging mode:

```
        node --inspect file.js # runs immediately
        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

```
        // The null primitive is typically used to describe
        // the absence of an object...
        // Null
        null

        // Undefined
        // ... whereas undefined is the absence
        // of a defined value.
        // Any variable initialized without a value will be undefined.
        // Any expression that attempts access of a non-existent
        property on an object will result in undefined.
        // A function without a return statement will return undefined.
        // undefined

        // Number
        // The Number type is double-precision floating-point format.
        // It allows both integers and decimals but
```

```
        // has an integer range of $-2^53-1$ to $2^53-1$.
        1, 1.5, -1e4, NaN

        // BigInt
        // The BigInt type has no upper/lower limit on integers.
        1n, 9007199254740993n

        // String
        'str', "str", `str ${var}`

        // Boolean
        true, false

        // Symbol
        // Symbols can be used as unique identifier keys in objects.
        //The Symbol.for method creates/gets a global symbol.
        Symbol('description'), Symbol.for('namespace')...


```

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

```
        // this refers to the object on which the function was called,
        // not the object the function was assigned to
        const obj = { id: 999, fn: function () { console.log(this.id) } }
        const obj2 = { id: 2, fn: obj.fn }
        obj2.fn() // prints 2
        obj.fn() // prints 999

        // Functions have a call method that can be used
        // to set their this context. See
        /**
        * Calls a method of an object,
```

```
        * substituting another object for the current object.
        * @param thisArg The object to be used as the current object.
        * @param argArray A list of arguments to be passed to the method.
        */
        call(this: Function, thisArg: any, ...argArray: any[]): any;

        function fn() { console.log(this.id) }
        const obj = { id: 999 }
        const obj2 = { id: 2 }
        fn.call(obj2) // prints 2
        fn.call(obj) // prints 999

        /*
        *Lambda functions do not have their own this context.
        * Wwhen this is *referenced inside a function,
        * it refers to the this of the
        *nearest parent non-lambda function.
        */
        function fn() {
            return (offset) => {
                console.log(this.id + offset)
            }
        }
        const obj = { id: 999 }
        const offsetter = fn.call(thisArg = obj)
        console.log(typeof(offsetter)); // function
        offsetter(1) // 1000

```

Lambda functions do not have a prototype.

3.2 Prototypal Inheritance

3.2.1 Prototypal inheritance - functional

Create a prototype chain:

```
    const wolf = {
        howl: function () { console.log(this.name + ': awooooooooo') }
    }

    const dog = Object.create(wolf, {
        woof: { value: function() { console.log(this.name + ': woof') } }
    })
```

```
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: awooooooooo"
```

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:

```
function Wolf (name) {
    this.name = name
}

Wolf.prototype.howl = function () {
    console.log(this.name + ': awooooooooo')
}
```

Define a function to set up the inheritance chain:

```
function inherit (proto) {
    function ChainLink(){}
    ChainLink.prototype = proto
    return new ChainLink()
}
```

Define how to obtain a child object:

```
function Dog (name) {
    Wolf.call(this, name + ' the dog')
}

Dog.prototype = inherit(Wolf.prototype)

Dog.prototype.woof = function () {
    console.log(this.name + ': woof')
}

```

Create a child object():

```
const rufus = new Dog('Rufus')

rufus.woof() // prints "Rufus the dog: woof"
rufus.howl() // prints "Rufus the dog: awooooooooo"
```

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

```
function Dog (name) {
    Wolf.call(this, name + ' the dog')
}

Dog.prototype = Object.create(Wolf.prototype)

Dog.prototype.woof = function () {
    console.log(this.name + ': woof')
}
```

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

```
const util = require('util')

function Dog (name) {
    Wolf.call(this, name + ' the dog')
}

Dog.prototype.woof = function () {
    console.log(this.name + ': woof')
}

// sets the prototype of Dog.prototype to Wolf.prototype
util.inherits(Dog, Wolf)

```

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

```
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:

```
class Wolf {
    constructor (name) {
        this.name = name
    }
    howl () { console.log(this.name + ': awooooooooo') }
}

class Dog extends Wolf {
    constructor(name) {
        super(name + ' the dog')
    }
    woof () { console.log(this.name + ': woof') }
}

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.

```
function wolf (name) {
    const howl = () => {
        console.log(name + ': awooooooooo')
    }
    return { howl: howl }
}

function dog (name) {
    name = name + ' the dog'
    const woof = () => { console.log(name + ': woof') }
    return {
        ...wolf(name),
        woof: woof
    }
}
const rufus = dog('Rufus')

console.log(rufus)
// { howl: [Function: howl], woof: [Function: woof] }
rufus.woof()
```

```
    // "Rufus the dog: woof"
    rufus.howl()
    // "Rufus the dog: awooooooooo"
```

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.

```
    // imports

    if (require.main === module) {
       // ...
    } else {
        const myfunc = (str) => {
            // ...
        }
        module.exports = myfunc
    }
```

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.*

```
npm start

// or:
node index.js

// app starts
```

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

```
$node -p 'require("./index.js")'
[Function: myfunc]

$ node -p 'require("./index.js")("test")'
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.

```
node -e "fs.renameSync('./format.js', './format.mjs');"
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

```
// Default export
export default () => {
    console.log('Hi from the default export!');
};

// Named export `doStuff`
export const doStuff = () => {
    console.log('Doing stuff');
};
```

This is a static import:

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

Whereas this is a dynamic import:

```

// returns a Promise
import('./utils.mjs')
.then((module) => {
    module.default();
    // logs 'Hi from the default export!'
    module.doStuff();
    // logs 'Doing stuff'
});

// or
(async () => {
    const moduleSpecifier = './utils.mjs';
    const module = await import(moduleSpecifier)
    module.default();
    // logs 'Hi from the default export!'
    module.doStuff();
    // logs 'Doing stuff'
})();
```

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:

```
{
    "name": "my-package",
    "version": "1.0.0",
    "main": "index.js",
    "type": "module",
    //...
}

```

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) => {
```

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

5.3.3 Importing modules

With static imports, different import possibilities exist:

- Implicitly import a module's default export

```
        // the default export of the url module is assigned
        //  to the url reference.
        import url from 'url'

```

- Import a specific named export from a module

```
        import { realpath } from 'fs/promises'

```

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

```
        import * as format from './format.js'

```

  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:

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

```
    `import 'tap'`,
    '=>',
    await resolve('tap', import.meta.url)
    )

    // resolved to [...]/tap/dist/esm/index.js


```

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.

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

    // __filename holds the path of the file currently being executed
    readFile(__filename, (err, contents) => {
        if (err) {
            console.error(err)
            return
        }
        console.log(contents.toString())
    })
```

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

```
    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, print)
    readFile(mediumFile, print)
    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:

```
    readFile(bigFile, (err, contents) => {
        print(err, contents)
        readFile(mediumFile, (err, contents) => {
            print(err, contents)
            readFile(smallFile, print)
        })
    })
```

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:

```

    function myAsyncOperation (cb) {
        doSomethingAsynchronous((err, value) => { cb(err, value) })
    }

    myAsyncOperation(functionThatHandlesTheResult)

```

This is the same in promise form:

```
function myAsyncOperation () {
    return new Promise((resolve, reject) => {
        // doSomethingAsynchronous expects a callback
        doSomethingAsynchronous((err, value) => {
            if (err) reject(err)
            else resolve(value)
        })
    })
}

const promise = myAsyncOperation()
// next up: do something with promise


```

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

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

const promise = myAsyncOperation()

```

Promise success or failure are handled using then and catch:

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

```

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
```

```
    .then((contents) => {
        print(contents)
        return readFile(smallFile)
        })
    // returns itself
   .then(print)
   .catch(console.error)

```

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.

```

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

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

```

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:

```

    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)

```

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:

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

```

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

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

    run().catch(console.error)

```

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 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)
```

```
    }

function sleep(delay) {
        return new Promise(resolve => setTimeout(resolve, delay));
    }

const run = async () => {
        // option 1, using then()
        opA(print).then(() => {
            // opC needs to called inside this!!
            opB(print).then(() => {
                opC(print)}
            )
        });

        await sleep(3000);

        // option 2, using await
        await opA(print);
        await opB(print);
        await opC(print);
    }

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:

```

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

    run().catch(console.error)

```

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
```

```
        const opB = async (cb) => {
            await setTimeoutPromise(() => {
                cb(null, 'B')
            }, 250)
        }

        const opC = async (cb) => {
            await setTimeoutPromise(() => {
                cb(null, 'C')
            }, 125)
        }

        const run = async () => {

            const A = opA(print);
            const B = opB(print);
            const C = opC(print);

            await A;
            await B;
            await C;

        }

        run()

```

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:

```

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

    const read = promisify((cb) => {
        let index = 0
        const print = (err, contents) => {
            index += 1
            if (err) {
                console.error(err)
```

```
                if (index === 3) cb()
                return
            }
            console.log(contents.toString())
            if (index === 3) cb()
        }
        readFile(bigFile, print)
        readFile(mediumFile, print)
        readFile(smallFile, print)
    })

    async function run () {
        await read()
        console.log('finished!')
    }

    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:

```

    import { setTimeout } from 'timers/promises'

    const ac = new AbortController()
    const { signal } = ac
    const timeout = setTimeout(1000, 'will NOT be logged', { signal })

    setImmediate(() => {
        ac.abort()
    })

    try {
        console.log(await timeout)
    } catch (err) {
        // ignore abort errors:
```

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

```

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 Event System

The EventEmitter constructor in the events module is the functional backbone of many Node core API's. For instance, HTTP and TCP servers are an event emitter, a TCP socket is an event emitter, HTTP request and response objects are event emitters. In this chapter, we'll explore how to create and consume EventEmitters.

7.1 Creating an Event Emitter

To be able to create an EventEmitter, either import it like this

```

    // the events module exports an EventEmitter constructor
    import { EventEmitter } from 'node:events';

    // this should work in .cjs files, but yields undefined
    // const { EventEmitter } = require('events')

```

or like so:

```

    // in modern node the events module is
    // the EventEmitter constructor, too :
    import EventEmitter from 'node:events';

    // see events.js:
    /*
    class EventEmitter<T extends EventMap<T> = DefaultEventMap> {
        constructor(options?: EventEmitterOptions);
        // ...
```

```
    }

    import internal = require("node:events");
    namespace EventEmitter {
        // Should just be `export { EventEmitter }`,
        // but that doesn't work in TypeScript 3.4
        export { internal as EventEmitter };
        // ...
    */

    // resp.
    const EventEmitter = require('events')

```

To create an EventEmitter instance, inherit from the EventEmitter class:

```

    class MyEmitter extends EventEmitter {}

    const myEmitter = new MyEmitter();
    myEmitter.on('event', () => {
        console.log('an event occurred!');
    });
    myEmitter.emit('event');

```

7.2 Emitting Events

To emit an event call the emit method:

```

    const { EventEmitter } = require('events')
    const myEmitter = new EventEmitter()
    // arguments: event namespace, 2/3: passed to listener
    myEmitter.emit('an-event', some, args)

```

7.3 Listening for Events

```
const { EventEmitter } = require('events')

const ee = new EventEmitter()
ee.on('close', () => { console.log('close event fired!') })
ee.emit('close')

```

Process additional arguments passed to emit like so:

```
ee.on('add', (a, b) => { console.log(a + b) }) // logs 13
ee.emit('add', 7, 6)

```

Listeners are called in the order they are registered. prepedListener is used to add a listener in top position.

```
const { EventEmitter } = require('events')
const ee = new EventEmitter()
ee.on('my-event', () => { console.log('1st') })
ee.on('my-event', () => { console.log('2nd') })
ee.prependListener('my-event', () => { console.log('really 1st') })
ee.emit('my-event')

```

To have a listener called at most once, use eventEmitter.once() instead of on().

```
const { EventEmitter } = require('events')
const ee = new EventEmitter()
ee.once('my-event', () => { console.log('my-event fired') })
ee.emit('my-event')
ee.emit('my-event')
ee.emit('my-event')

```

To remove listeners, use removeListener("eventName", listenerVar).
Call removeAllListeners("eventName") to remove them all.

```
1
2     const { EventEmitter } = require('events')
3     const ee = new EventEmitter()
4
5     const listener1 = () => { console.log('listener 1') }
6     const listener2 = () => { console.log('listener 2') }
7
8     ee.on('my-event', listener1)
9     ee.on('my-event', listener2)
10
11    setInterval(() => {
12        ee.emit('my-event')
13    }, 200)
14
15    setTimeout(() => {
16        ee.removeListener('my-event', listener1)
17    }, 500)
18
19    setTimeout(() => {
20        ee.removeListener('my-event', listener2)
21    }, 1100)
22
```

7.4 The error Event

Emitting an 'error' event on an event emitter will cause the event emitter to throw an exception if a listener for the 'error' event has not been registered.

Here a crash will not occur:

```
1
2     const { EventEmitter } = require('events')
3     const ee = new EventEmitter()
4
5     process.stdin.resume() // keep process alive
6
7     ee.on('error', (err) => {
8         console.log('got error:', err.message )
9     })
10
11    ee.emit('error', new Error('oh oh'))
12
```

7.5 Promise-Based Single Use Listener and AbortController

When awaiting an event, AbortController can be used as an escape hatch.

```

    import { once, EventEmitter } from 'events'
    import { setTimeout } from 'timers/promises'

    const uneventful = new EventEmitter()

    const ac = new AbortController()
    const { signal } = ac

    setTimeout(500).then(() => ac.abort())

    try {
        await once(uneventful, 'ping', { signal })
        console.log('pinged!')
    } catch (err) {
        // ignore abort errors:
        if (err.code !== 'ABORT_ERR') throw err
        console.log('canceled')
    }

```

8 Handling Errors

8.1 Native Error Constructors

Error is the native constructor for generating an error object. To create an error, call new Error and pass a string as a message:

```
    new Error('this is an error message')
```

There are six other native error constructors that inherit from the base Error constructor, these are:

```
    EvalError
    SyntaxError
```

```
    RangeError
    ReferenceError
    TypeError
    URIError
```

There's mainly two errors that are likely to be thrown in library or application code: RangeError and TypeError.

8.2 Custom Errors: Set error code on Error object

```
    function doTask (amount) {
        if (typeof amount !== 'number')
         throw new TypeError('amount must be a number')
        if (amount <= 0)
         throw new RangeError('amount must be greater than zero')
        if (amount % 2) {
            const err = Error('amount must be even')
            err.code = 'ERR_MUST_BE_EVEN'
            throw err
        }
        return amount / 2
    }
```

8.3 Custom Errors: Inheriting from Error

```
    class OddError extends Error {
        constructor (varName = '') {
            super(varName + ' must be even')
        }
        get name () { return 'OddError' }
    }

    // can additionally add a code
    class OddError extends Error {
        constructor (varName = '') {
            super(varName + ' must be even')
            this.code = 'ERR_MUST_BE_EVEN'
        }
        get name () {
            return 'OddError [' + this.code + ']'
```

```
        }
    }
```

8.4 Try/Catch

We can separately handle different kinds of errors like so:

```
    try {
        const result = doTask(4)
        console.log('result', result)
    } catch (err) {
        if (err instanceof TypeError) {
            console.error('wrong type')
        } else if (err instanceof RangeError) {
            console.error('out of range')
        } else if (err instanceof OddError) {
            console.error('cannot be odd')
        } else {
            console.error('Unknown error', err)
        }
    }
```

However, it is more reliable to throw and check for error codes:

```
    function codify (err, code) {
        err.code = code
        return err
    }

    function doTask (amount) {
        if (typeof amount !== 'number') throw codify(
        new TypeError('amount must be a number'),
        'ERR_AMOUNT_MUST_BE_NUMBER'
        )
        if (amount <= 0) throw codify(
        new RangeError('amount must be greater than zero'),
        'ERR_AMOUNT_MUST_EXCEED_ZERO'
        )
        if (amount % 2) throw new OddError('amount')
        return amount/2
    }

```

```
    try {
        const result = doTask(4)
        result()
        console.log('result', result)
    } catch (err) {
        if (err.code === 'ERR_AMOUNT_MUST_BE_NUMBER') {
            console.error('wrong type')
        } else if (err.code === 'ERRO_AMOUNT_MUST_EXCEED_ZERO') {
            console.error('out of range')
        } else if (err.code === 'ERR_MUST_BE_EVEN') {
            console.error('cannot be odd')
        } else {
            console.error('Unknown error', err)
        }
    }
```

It's important to keep in mind that try/catch cannot catch errors that are thrown in a callback function that is called at some later point. Consider the following:

```
    // WARNING: NEVER DO THIS:
    try  {
        setTimeout(() => {
            const result = doTask(3)
            console.log('result', result)
        }, 100)
    } catch (err) {
        // ...
    }
```

An easy fix is to move the try/catch into the body of the callback function:

```
    setTimeout(() => {
        try  {
            const result = doTask(3)
            console.log('result', result)
        } catch (err) {
            // ...
        }
    }, 100)
```

8.5 Rejected Promises

Assuming a function that accepts a Promise:

```
function doTask (amount) {
    return new Promise((resolve, reject) => {
        if (typeof amount !== 'number') {
            reject(new TypeError('amount must be a number'))
            return
        }
        if (amount <= 0) {
            //
        }
        if (amount % 2) {
            //
        }
        resolve(amount/2)
    })
}
```

to catch rejections we have to use a catch clause:

```
doTask(3)
.then((result) => {
    //
})
.catch((err) => {
    //
})
```

When the throw appears inside a promise handler, that will not be an exception, that is, it won't be an error that is synchronous. Instead it will be a rejection: The then or catch handler will return a *new promise* that rejects as a result of a throw within the handler.

Example:

```
doTask(4)
.then((result) => {
    throw Error('spanner in the works')
})
.catch((err) => {
    //
})
```

Here the catch clause does not run, and an error is thrown.

8.6 Async Try/Catch

The same try/catch works for async/await, as async/await is nothing but syntactic sugar around Promise.

For example:

```
async function run () {
    try {
        //
    } catch (err) {
        //
    }
}
```

If a throw occurs in an async function, the returned promise rejects. This means that with async/await, we can simply throw again. As a result, the code looks exaxtly like synchronous code, apart from the async keyword itself:

```
async function doTask (amount) {
    if (typeof amount !== 'number')
     throw new TypeError('amount must be a number')
    if (amount <= 0)
     throw new RangeError('amount must be greater than zero')
    if (amount % 2) throw new OddError('amount')
    return amount/2
}
```

In consequence, async/await can conveniently be nested:

```
async function doTask (amount) {
    if //
    if // ...
    // fetch something from a server
    const result = await asyncFetchResult(amount)
    return result
}
```

Here, if the promise returned from asyncFetchResult rejects, this will cause the promise returned from doTask to reject. Thus, one catch block can be used to handle all cases.

8.7 Error Propagation

Error propagation is where, instead of handling the error, we make it the responsibility of the caller instead. Here is an example handling known errors but propagating unknown ones:

```

    // custom error definitions ....

    async function doTask (amount) {
        if (/* ...  */) throw new TypeError(// ...
        if (/* ...  */) throw new RangeError(// ...
        if (/* ...  */) throw new OddError(// ...
        return amount/2
    }

    async function run () {
        try {
            const result = await doTask(4)
            console.log('result', result)
        } catch (err) {
            if // ...
            } else if // ...
            } else if // ...
            // unknown error
            } else {
                throw err
            }
        }
    }
    // catch remaining errors
    run().catch((err) => { console.error('Error caught', err) })
```

Here is the equivalent example using a callback:

```

    // now takes a callback
    function doTask (amount, cb) {
        if (/* ... */ {
            cb( // .... throw some Error
            return
        }
        if (/* ... */ {
            cb( // .... throw some Error
```

```
            return
        }
        if (/* ... */ {
            cb(Error('some other error'))
            return
        }
        // first argument (Error) is null, pass result as second!
        cb(null, amount/2)
    }

    // 1. passes an error-first callback to doTask()
    // 2. later, calls its own caller using the passed-in argument cb
    // beware: cb is not the callback passed to doTask!
    function run (cb) {
        // callback passed to doTask() is created by run()!
        doTask(4, (err, result) => {
            // if error, check code
            // if not a known one pass on as-is
            // first argument (Error) is not null
            if (err) {
                if (err.code === // ...
                    cb(Error('wrong type'))
                } else if (err.code === // ...
                    cb(Error('out of range'))
                } else if // ...
                    cb(Error('cannot be odd'))
                } else {
                    cb(err)
                }
                return
            }

            // do some real stuff with result
            console.log('result', result)
        })
    }

    run((err) => {
        if (err) console.error('Error caught', err);
    })


```

9 Buffers

A buffer object is both an instance of Buffer and an instance (at the second degree) of Uint8Array (8 bytes of unsigned integers between 0-255; subclass of non-instantiable TypedArray). In consequence, methods from Uint8Array as well as those from Buffer are available.

To safely create a buffer (meaning, zeroed-out bytes), we use Buffer.alloc(num_bytes).

Buffers can conveniently be converted into/from strings:

```
\\ not required, but recommended
const { Buffer } = require('node:buffer');

const buffer = Buffer.from('hello world')
buffer

// <Buffer 68 65 6c 6c 6f 20 77 6f 72 6c 64>

```

9.1 Buffer creation

The default character set is UTF-8, which uses up to 4 bytes:

```
// U+1F440 not displayed by tex
// length of string: ?!?
// https://mathiasbynens.be/notes/javascript-unicode
console.log('<EYES>'.length)
// 2

// utf-8 encoding uses 8 bytes
console.log(Buffer.from('<EYES>').length)
// 4
Buffer.from('U+1F440')
<Buffer f0 9f 91 80>
```

But a different character set can be specified:

```
Buffer.from("U+00fc")
<Buffer c3 bc>

Buffer.from("U+00fc", "ascii")
<Buffer fc>

Buffer.from("U+00fc", "utf16le")
```

```
8     <Buffer fc 00>
9
```

9.2 Character encodings

The *character encodings* currently supported by Node.js are the following:

- 'utf8' (alias: 'utf-8')
- 'utf16le' (alias: 'utf-16le'): each character in the string will be encoded using either 2 or 4 bytes. Node.js only supports the little-endian variant of UTF-16.
- 'latin1': = ISO-8859-1. This character encoding only supports the Unicode characters from U+0000 to U+00FF. Each character is encoded using a single byte.

Converting a *Buffer into a string* using one of the above is referred to as *decoding*, and converting *a string into a Buffer* is referred to as *encoding*.

The following legacy character encodings are also supported:

- 'ascii': For 7-bit ASCII data only. When encoding a string into a Buffer, this is equivalent to using 'latin1'. When decoding a Buffer into a string, using this encoding will additionally unset the highest bit of each byte before decoding as 'latin1'. Generally, there should be no reason to use this encoding, as 'utf8' (or, if the data is known to always be ASCII-only, 'latin1') will be a better choice when encoding or decoding ASCII-only text. It is only provided for legacy compatibility.
- 'binary': Alias for 'latin1'. The name of this encoding can be very misleading, as all of the encodings listed here convert between strings and binary data. For converting between strings and Buffers, typically 'utf8' is the right choice.
- 'ucs2', 'ucs-2': Aliases of 'utf16le'. UCS-2 used to refer to a variant of UTF-16 that did not support characters that had code points larger than U+FFFF. In Node.js, these code points are always supported.

9.3 Binary-to-text encodings

Node.js also supports the following *binary-to-text* encodings. For binary-to-text encodings, the naming convention is reversed: Converting a *Buffer into a string* is typically referred to as *encoding*, and converting *a string into a Buffer* as *decoding*.

- 'base64': Base64 encoding. Whitespace characters such as spaces, tabs, and new lines contained within the base64-encoded string are ignored.
- 'base64url': base64url encoding as specified in RFC 4648, Section 5.When encoding a Buffer to a string, this encoding will omit padding.

- 'hex': Encode each byte as two hexadecimal characters. Data truncation may occur when decoding strings that do not exclusively consist of an even number of hexadecimal characters. See below for an example.

Example:

```
// utf-8 encoding of ASCII STRING
Buffer.from("8J+RgA==")
<Buffer 38 4a 2b 52 67 41 3d 3d>

// EYES is 8J+RgA== in base64
// utf-8 encoding of base64 decoding
Buffer.from("8J+RgA==", "base64")
<Buffer f0 9f 91 80>

// same as
Buffer.from('<EYES>')
<Buffer f0 9f 91 80>
```

9.4 Decoding to String

When converting to string, we can also specify an encoding:

```
const buffer = Buffer.from('<EYES>')
console.log(buffer)
<Buffer f0 9f 91 80>

console.log(buffer.toString()) // default utf-8
<EYES>

console.log(buffer.toString('hex'))
f09f9180

console.log(buffer.toString('base64'))
8J+RgA==console.log(buffer.toString(')) // prints 8J+RgA==
```

9.5 Using StringDecoder

The UTF8 encoding format has between 1 and 4 bytes to represent each character, if for any reason one or more bytes is truncated from a character this will result in encoding errors. So in situations where we have multiple buffers that might split characters across a byte boundar y the Node core string_decoder module should be used.

Calling decoder.write will output a character only when all of the bytes representing that character have been written to the decoder:

```
const { StringDecoder } = require('string_decoder')

const frag1 = Buffer.from('f09f', 'hex')
const frag2 = Buffer.from('9180', 'hex')

console.log(frag1.toString())
<question_mark>

console.log(frag2.toString())
 <question_mark>

const decoder = new StringDecoder()

console.log(decoder.write(frag1)) // prints nothing
console.log(decoder.write(frag2)) // prints <EYES>
<EYES>

```

9.6 JSON Serializing and Deserializing Buffers

When JSON.stringify encounters any object it will attempt to call a toJSON method on that object if it exists. Buffer instances have a toJSON method which returns a plain JavaScript object in order to represent the buffer in a JSON-friendly way:

```
buffer
<Buffer f0 9f 91 80>

buffer.toJSON()
{ type: 'Buffer', data: [ 240, 159, 145, 128 ] }

JSON.stringify(buffer)
'{"type":"Buffer","data":[240,159,145,128]}'
```

When deserializing, JSON.parse will only turn that JSON representation of the buffer into a plain JavaScript object, to turn it into an object the data array must be passed to Buffer.from:

```
const buffer = Buffer.from('<EYES>')
const json = JSON.stringify(buffer)
```

```
3
4     const parsed = JSON.parse(json)
5     console.log(parsed)
6     { type: 'Buffer', data: [ 240, 159, 145, 128 ] }
7
8     console.log(Buffer.from(parsed.data))
9     <Buffer f0 9f 91 80>
```

When an array of numbers is passed to Buffer.from they are converted to a buffer with byte values corresponding to those numbers.

9.7 Example

```
1     const str = 'buffers are neat'
2
3     // <Buffer 62 75 66 66 65 72 73 20 61 72 65 20 6e 65 61 74>
4     let y = Buffer.from(str)
5     // 'YnVmZmVycyBhcmUgbmVhdA=='
6     let z = y.toString('base64')
7
8     // 1liner
9     // const base64 = Buffer.from(str).toString('base64')
10
11    assert.equal(z, Buffer.from([
12    89,110,86,109,90,109,86,121,99,
13    121,66,104,99,109,85,103,98,109,
14    86,104,100,65,61,61]))
```

10 Streams

10.1 Stream Types

Constructors exposed by the base *stream* module are:

```
1     Stream
2     Readable
3     Writable
4     Duplex
5     Transform
6     PassThrough
```

The Stream constructor inherits from the *EventEmitter* constructor from the *events* module.

Events emitted by various Stream implementations are:

```
    data
    end
    finish
    close
    error
```

10.2 Stream Modes

The mode of a stream is determined by its *objectMode* option passed when the stream is instantiated, the default being binary. *Binary mode* streams only read or write *Buffer* instances. *object mode* streams read or write *objects and primitives* (strings, numbers) except null.

10.3 Readable Streams

To create, using fs' *createReadStream* constructor:

```
'use strict'
const fs = require('fs')

// file executing the code
const readable = fs.createReadStream(__filename)

readable.on('data', (data) => { console.log(' got data', data) })
readable.on('end', () => { console.log(' finished reading') })

```

Readable streams are usually connected to an I/O layer via a C-binding, but we can create a contrived readable stream ourselves using the Readable constructor:

```
    'use strict'
    const { Readable } = require('stream')

    const createReadStream = () => {
        const data = ['some', 'data', 'to', 'read']
        // pass an options object with a read method
        return new Readable({
            // ca pass size (how many bytes to read),
            // highWaterMark (default 16kb)
```

```
            read () {
                // this points to readable stream instance
                if (data.length === 0) this.push(null)
                else this.push(data.shift())
            }
        })
    }

    const readable = createReadStream()

    readable.on('data', (data) => { console.log('got data', data) })
    readable.on('end', () => { console.log('finished reading') })
```

When each data event is emitted it receives a string instead of a buffer. However, the default stream mode is objectMode: false, meaning Buffer.

Thus, a string is pushed to the Readable stream. That stream is then converted to a buffer. Finally, it is decoded to string using UTF8.

When creating a readable stream without the intention of using buffers, we can instead set objectMode to true:

```
    'use strict'
    const { Readable } = require('stream')

    const createReadStream = () => {
        const data = ['some', 'data', 'to', 'read']
        return new Readable({
            objectMode: true,
            read () {
                if (data.length === 0) this.push(null)
                else this.push(data.pop())
            }
        })
    }

    const readable = createReadStream()

    readable.on('data', (data) => { console.log('got data', data) })
    readable.on('end', () => { console.log('finished reading') })
```

This time the string is being sent from the readable stream without converting to a buffer first.

Using *Readable.from*, streams can be created from iterable data structures, like arrays:

```
    'use strict'
    const { Readable } = require('stream')

    // objectMode is true by default
    const readable = Readable.from(['some', 'data', 'to', 'read'])

    readable.on('data', (data) => { console.log('got data', data) })
    readable.on('end', () => { console.log('finished reading') })

```

10.4 Writable Streams

To create:

```
'use strict'
const fs = require('fs')

const writable = fs.createWriteStream('./out')
writable.on('finish', () => { console.log('finished writing') })

// strings are converted to buffers
writable.write('A\n')
writable.write('B\n')
writable.write('C\n')
writable.end('nothing more to write')
```

Like Readable streams, Writable streams are mostly useful for I/O, which means integrating a writable stream with a native C-binding, but we can likewise create a contrived write stream example:

```
'use strict'
const { Writable } = require('stream')

const createWriteStream = (data) => {
    return new Writable({
        write (chunk, enc, next) {
            data.push(chunk)
            // we are ready for the next piece of data
            next()
        }
    })
```

```
}
const data = []

const writable = createWriteStream(data)
writable.on('finish', () => { console.log('finished writing', data) })

writable.write('A\n')
writable.write('B\n')
writable.write('C\n')
writable.end('nothing more to write')

```

The point of a next callback function is to allow for asynchronous operations within the write. This is essential for performing asynchronous I/O.

Note again, as with readable streams, the default objectMode option is false, so each string written to our writable stream instance is converted to a buffer before it becomes the chunk argument passed to the write option function.

If we are dealing with strings or Buffers only, we can avoid the unnecessary conversion by setting the decodeStrings option to false:

```
const createWriteStream = (data) => {
    return new Writable({
        decodeStrings: false,
        write (chunk, enc, next) {
            data.push(chunk)
            next()
        }
    })
}
```

Now trying to pass anything else (e.g., a number) will result in an error because we're attempting to write a JavaScript value that isn't a string to a binary stream.

If instead we want to support strings and any other JavaScript value, we can instead set objectMode to true to create an object-mode writable stream:

```
'use strict'
const { Writable } = require('stream')

const createWriteStream = (data) => {
    return new Writable({
        objectMode: true,
        write (chunk, enc, next) {
            data.push(chunk)
```

```
            next()
        }
    })
}
const data = []

const writable = createWriteStream(data)
writable.on('finish', () => { console.log('finished writing', data) })
writable.write('A\n')
writable.write(1)
writable.end('nothing more to write')
```

10.5 Readable-Writable Streams

In addition to the Readable and Writable stream constructors there are three more core stream constructors that have both readable and writable interfaces: Duplex, Transform, and PassThrough.

10.5.1 Duplex

With a Duplex stream, both read and write methods are implemented but there doesn't have to be a causal relationship between them. Just because something is written to a Duplex stream doesn't necessarily mean that it will result in any change to what can be read from the stream, although it might.

A TCP network socket is a great example of a Duplex stream:

```
    'use strict'
    const net = require('net')

    net.createServer((socket) => {
        const interval = setInterval(() => {
            // writable side of the stream
            socket.write('beat')
        }, 1000)
        // readable side of the stream
        socket.on('data', (data) => {
            // writable side of the stream
            socket.write(data.toString().toUpperCase())
        })
        // readable side of the stream
        socket.on('end', () => { clearInterval(interval) })
    }).listen(3000)
```

The client socket is also a Duplex stream:

```
'use strict'
const net = require('net')

const socket = net.connect(3000)

socket.on('data', (data) => {
    console.log('got data:', data.toString())
})

socket.write('hello')

setTimeout(() => {
    socket.write('all done')
    setTimeout(() => {
        socket.end()
    }, 250)
}, 3250)
```

10.5.2 Transform

The Transform constructor inherits from the Duplex constructor. Transform streams are duplex streams with an additional constraint applied to enforce a causal relationship between the read and write interfaces. A good example is compression:

```

'use strict'
const { createGzip } = require('zlib')

const transform = createGzip()

transform.on('data', (data) => {
    console.log('got gzip data', data.toString('base64'))
})
transform.write('first')
setTimeout(() => {
    transform.end('second')
}, 500)
```

The way that Transform streams create this causal relationship is through how a transform stream is created. Instead of supplying read and write options functions, a transform option is passed to the Transform constructor:

```
'use strict'
const { Transform } = require('stream')
const { scrypt } = require('crypto')

const createTransformStream = () => {
    return new Transform({
        decodeStrings: false,
        encoding: 'hex',
        // same signature as write but
        // the next function can be passed a second argument
        // which should be the result of applying some kind
        // of transform operation to the incoming chunk.
        transform (chunk, enc, next) {
            scrypt(chunk, 'a-salt', 32, (err, key) => {
                if (err) {
                    // emit an error event
                    next(err)
                    return
                }
                // null indicates no error,
                // and data event is emitted from the readable side
                next(null, key)
            })
        }
    })
}
const transform = createTransformStream()
transform.on('data', (data) => {
    console.log('got data:', data)
})
transform.write('A\n')
transform.write('B\n')
transform.write('C\n')
transform.end('nothing more to write')

```

10.5.3 PassThrough

The PassThrough constructor inherits from the Transform constructor. It's essentially a transform stream where no transform is applied. It implements the identity function, that is, it's a useful placeholder when a transform stream is expected but no transform is desired.

10.5.4 Determining End-of-Stream

As we discussed earlier, there are at least four ways for a stream to potentially become inoperative: close event, error event, finish event, and end event.

We often need to know when a stream has closed so that resources can be deallocated, otherwise memory leaks become likely.

Instead of listening to all four events, the stream.finished utility function provides a simplified way to do this:

```
'use strict'
const net = require('net')
const { finished } = require('stream')

net.createServer((socket) => {
    const interval = setInterval(() => {
        socket.write('beat')
    }, 1000)
    socket.on('data', (data) => {
        socket.write(data.toString().toUpperCase())
    })
    // The stream (socket) is passed to finished as the first argument
    // the second argument is a callback for when the stream ends
    // for any reason
    finished(socket, (err) => {
        // first argument of the callback is a potential error object
        // If the stream were to emit an error event the callback would
        // be called with the error object emitted by that event.
        if (err) {
            console.error('there was a socket error', err)
        }
        clearInterval(interval)
    })
}).listen(3000)

```

10.5.5 Piping Streams

Let's adapt the TCP client server from the "Readable-Writable Streams" page to use the pipe method. Here is the client from earlier:

```
'use strict'
const net = require('net')

const socket = net.connect(3000)

socket.on('data', (data) => {
    console.log('got data:', data.toString())
})

socket.write('hello')
setTimeout(() => {
    socket.write('all done')
    setTimeout(() => {
        socket.end()
    }, 250)
}, 3250)
```

We'll replace the data event listener with a pipe:

```
'use strict'
const net = require('net')

const socket = net.connect(3000)

socket.pipe(process.stdout)

socket.write('hello')
setTimeout(() => {
    socket.write('all done')
    setTimeout(() => {
        socket.end()
    }, 250)
}, 3250)
```

The pipe method exists on Readable streams (recall socket is a Duplex stream instance and that Duplex inherits from Readable), and is passed a Writable stream (or a stream with Writable capabilities). Internally, the pipe method sets up a data listener on the readable stream and automatically writes to the writable stream as data becomes available.

Since pipe returns the stream passed to it, it is possible to chain pipe calls together: streamA.pipe(streamB).pipe(streamC). This is a commonly observed practice, but it's also bad practice to create pipelines this way. If a stream in the middle fails or closes for any reason, the other streams in the pipeline will not automatically close. This can create severe memory leaks and other bugs.

The correct way to pipe multiple streams is to use the stream.pipeline utility function.

Combining the above Transform stream and the TCP server to create a pipeline of streams:

```
    'use strict'
    const net = require('net')
    const { Transform, pipeline } = require('stream')
    const { scrypt } = require('crypto')
    const createTransformStream = () => {
        return new Transform({
            decodeStrings: false,
            encoding: 'hex',
            transform (chunk, enc, next) {
                scrypt(chunk, 'a-salt', 32, (err, key) => {
                    if (err) {
                        next(err)
                        return
                    }
                    next(null, key)
                })
            }
        })
    }

    net.createServer((socket) => {
        const transform = createTransformStream()
        const interval = setInterval(() => {
            socket.write('beat')
        }, 1000)
        pipeline(socket, transform, socket, (err) => {
            if (err) {
                console.error('there was a socket error', err)
            }
            clearInterval(interval)
        })
    }).listen(3000)

```

The pipeline command will call pipe on every stream passed to it, and will allow a function to be passed as the final function. Note how we removed the finished utility method. This is because the final function passed to the pipeline function will be called if any of the streams in the pipeline close or fail for any reason.

11 Interacting with the File System

11.1 File Paths

```
    // absolute file paths
    console.log('current filename', __filename)
    console.log('current dirname', __dirname)
```

The path.join method generates platform-independent paths.

```
    'use strict'
    const { join } = require('path')
    console.log('out file:', join(__dirname, 'out.txt'))
```

Alongside path.join the other path builders are:

```

    // Given two absolute paths, calculates the relative path between them.
    path.relative

    // e.g. path.resolve('/foo', 'bar', 'baz') yields '/foo/bar/baz',
    // which is akin to executing cd /foo then cd bar then cd baz
    path.resolve

    // Resolves .. and . dot in paths and strips extra slashes, for
    // instance path.normalize('/foo/../bar//baz') returns '/bar/baz
    path.normalize

    path.format
    // Builds a string from an object. The object shape 'that path.format
    // accepts, corresponds to the object returned from path.parse
```

The path deconstructors are path.parse, path.extname, path.dirname and path.basename.

```
    'use strict'
    const { parse, basename, dirname, extname } = require('path')
    console.log('filename parsed:', parse(__filename))
```

```
    console.log('filename basename:', basename(__filename))
    console.log('filename dirname:', dirname(__filename))
    console.log('filename extname:', extname(__filename))
```

11.2 Reading and Writing

The fs module has lower level and higher level APIs. The lower level API's closely mirror POSIX system calls. For instance, fs.open opens and possibly creates a file and provides a file descriptor handle, taking same options as the POSIX open command.

The higher level methods for reading and writing are provided in four abstraction types: Synchronous, callback-based, promise-based, and stream-based.

11.2.1 Synchronous

E.g.

```
'use strict'
const { readFileSync } = require('fs')

const contents = readFileSync(__filename)
// or set encoding
const contents = readFileSync(__filename, {encoding: 'utf8'})

console.log(contents)
```

The fs.writeFileSync function takes a path and a string or buffer and blocks the process until the file has been completely written:

```
'use strict'
const { join } = require('path')

const { readFileSync, writeFileSync } = require('fs')
const contents = readFileSync(__filename, {encoding: 'utf8'})
writeFileSync(join(__dirname, 'out.txt'), contents.toUpperCase())
// or
writeFileSync(join(__dirname, 'out.txt'), contents.toUpperCase(), {
    flag: 'a'
})

```

11.2.2 Callback-based

Read:

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

readFile(__filename, {encoding: 'utf8'}, (err, contents) => {
    if (err) {
        console.error(err)
        return
    }
    console.log(contents)
})
```

Read and write, asynchronously each:

```
'use strict'
const { join } = require('path')
const { readFile, writeFile } = require('fs')

readFile(__filename, {encoding: 'utf8'}, (err, contents) => {
    if (err) {
        console.error(err)
        return
    }
    const out = join(__dirname, 'out.txt')
    writeFile(out, contents.toUpperCase(), (err) => {
        if (err) { console.error(err) }
    })
})
```

11.2.3 Promise-based

The fs/promises API provides most of the same asynchronous methods that are available on fs, but the methods return promises instead of accepting callbacks.

So instead of loading readFile and writeFile like so:

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

We can load the promise-based versions like so:

```
const { readFile, writeFile } = require('fs/promises')
```

It's also possible to load fs/promises with

```
require('fs').promises
```

, which is backwards compatible with legacy Node versions (v12 and v10), but fs/promises supersedes fs.promises and aligns with other more recent API additions (such as stream/promises and timers/promises).

Let's look at the same reading and writing example using fs/promises and using async/await to resolve the promises:

```
'use strict'
const { join } = require('path')
const { readFile, writeFile } = require('fs/promises')

async function run () {
    const contents = await readFile(__filename, {encoding: 'utf8'})
    const out = join(__dirname, 'out.txt')
    await writeFile(out, contents.toUpperCase())
}

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

11.2.4 File Streams

The fs module has fs.createReadStream and fs.createWriteStream methods which allow us to read and write files in chunks. Streams are ideal when handling very large files that can be processed incrementally.

Let's start by simply copying the file:

```
'use strict'
const { pipeline } = require('stream')
const { join } = require('path')
const { createReadStream, createWriteStream } = require('fs')

pipeline(
createReadStream(__filename),
createWriteStream(join(__dirname, 'out.txt')),
(err) => {
    if (err) {
```

```
            console.error(err)
            return
        }
        console.log('finished writing')
    }
    )
```

To reproduce the read, upper-case, write scenario we created in the previous section, we will need a transform stream to upper-case the content:

```
    'use strict'
    const { pipeline } = require('stream')
    const { join } = require('path')
    const { createReadStream, createWriteStream } = require('fs')
    const { Transform } = require('stream')
    const createUppercaseStream = () => {
        return new Transform({
            transform (chunk, enc, next) {
                const uppercased = chunk.toString().toUpperCase()
                next(null, uppercased)
            }
        })
    }

    pipeline(
    createReadStream(__filename),
    createUppercaseStream(),
    createWriteStream(join(__dirname, 'out.txt')),
    (err) => {
        if (err) {
            console.error(err)
            return
        }
        console.log('finished writing')
    }
    )
```

11.2.5 Reading Directories

Directories are a special type of file, which hold a catalog of files. Similar to files the fs module provides multiple ways to read a directory: synchronous, callback-based, promise-based, and an async iterable that inherits from fs.Dir.

Let's look at synchronous, callback-based and promise-based at the same time:

```
'use strict'
const { readdirSync, readdir } = require('fs')
const { readdir: readdirProm } = require('fs/promises')

// synchronous method may throw so wrap in try/catch
try {
    console.log('sync', readdirSync(__dirname))
} catch (err) {
    console.error(err)
}

// callback
readdir(__dirname, (err, files) => {
    if (err) {
        console.error(err)
        return
    }
    console.log('callback', files)
})

async function run () {
    const files = await readdirProm(__dirname)
    console.log('promise', files)
}

// If readdirProm does reject, run will likewise reject
// This is why a catch handler is attached
run().catch((err) => {
    console.error(err)
})

```

Example: streaming directory contents over HTTP in JSON format:

```
'use strict'
const { createServer } = require('http')
const { Readable, Transform, pipeline } = require('stream')
const { opendir } = require('fs')


const createEntryStream = () => {
    let syntax = '[\n'
```

```
9
10          return new Transform({
11              // true because dirStream is an object stream
12              writableObjectMode: true,
13              // false because res is a binary stream
14              readableObjectMode: false,
15              transform (entry, enc, next) {
16                  next(null, `${syntax} "${entry.name}"`)
17                  syntax = ',\n'
18              },
19              final (cb) {
20                  this.push('\n]\n')
21                  cb()
22              }
23          })
24      }
25
26      createServer((req, res) => {
27          if (req.url !== '/') {
28              res.statusCode = 404
29              res.end('Not Found')
30              return
31          }
32
33          opendir(__dirname, (err, dir) => {
34              if (err) {
35                  res.statusCode = 500
36                  res.end('Server Error')
37                  return
38              }
39              // dir not a stream, but an async iterable
40              const dirStream = Readable.from(dir)
41              const entryStream = createEntryStream()
42              res.setHeader('Content-Type', 'application/json')
43              // pipeline from dirStream to entryStream to res
44              // passing a final callback to pipeline
45              pipeline(dirStream, entryStream, res, (err) => {
46                  if (err) console.error(err)
47              })
48          })
49      }).listen(3000)
```

11.3 File Metadata

Metadata about files can be obtained with the following methods:

```
fs.stat
fs.statSync
fs/promises stat

fs.lstat
fs.lstatSync
fs/promises lstat
```

The only difference between the stat and lstat methods is that stat follows symbolic links, and lstat will get metadata for symbolic links instead of following them.

These methods return an fs.Stat instance which has a variety of properties and methods for looking up metadata about a file.

Let's start by reading the current working directory and finding out whether each entry is a directory or not.

```
'use strict'
const { readdirSync, statSync } = require('fs')

const files = readdirSync('.')

for (const name of files) {
    const stat = statSync(name)
    const typeLabel = stat.isDirectory() ? 'dir: ' : 'file: '
    console.log(typeLabel, name)
}

```

Let's extend our example with time stats. There are four stats available for files:

- Access time
- Change time
- Modified time
- Birth time

The difference between change time and modified time, is modified time only applies to writes (although it can be manipulated by fs.utime), whereas change time applies to writes and any status changes such as changing permissions or ownership.

With default options, the time stats are offered in two formats, one is a Date object and the other is milliseconds since the epoch. We'll use the Date objects and convert them to locale strings.

Let's update our code to output the four different time stats for each file:

```
'use strict'
const { readdirSync, statSync } = require('fs')

const files = readdirSync('.')

for (const name of files) {
    const stat = statSync(name)
    const typeLabel = stat.isDirectory() ? 'dir: ' : 'file: '
    const { atime, birthtime, ctime, mtime } = stat
    console.group(typeLabel, name)
    console.log('atime:', atime.toLocaleString())
    console.log('ctime:', ctime.toLocaleString())
    console.log('mtime:', mtime.toLocaleString())
    console.log('birthtime:', birthtime.toLocaleString())
    console.groupEnd()
    console.log()
}
```

11.4 Watching

The fs.watch method is provided by Node core to tap into file system events. It is, however, fairly low level and not the most friendly of APIs. It's worth considering the ecosystem library, chokidar for file watching needs as it provides a friendlier high level API.

Example use:

```
'use strict'
const { watch } = require('fs')

watch('.', (evt, filename) => {
    console.log(evt, filename)
})
```

12 Process and Operating System

12.1 STDIO

Example:

```
    'use strict'
    console.log('initialized')
    const { Transform } = require('stream')

    const createUppercaseStream = () => {
        return new Transform({
            transform (chunk, enc, next) {
                const uppercased = chunk.toString().toUpperCase()
                next(null, uppercased)
            }
        })
    }

    const uppercase = createUppercaseStream()

    process.stdin.pipe(uppercase).pipe(process.stdout)
```

12.2 Process Info

Among others:

```
    'use strict'

    console.log('Current Directory', process.cwd())
    console.log('Process Platform', process.platform)
    console.log('Process ID', process.pid)
```

To get the environment variables we can use process.env, and query for specific keys: To set environment variables, create a new key.

```
    process.env.MODULEPATH
    /etc/modulefiles:/usr/share/modulefiles:/usr/share/modulefiles/Linux:
    /usr/share/modulefiles/Core:/usr/share/lmod/lmod/modulefiles/Core

    process.env.FOO='my env var'
```

12.3 Process Stats

The process object has methods which allow us to query resource usage. We're going to look at the process.uptime(), process.cpuUsage and process.memoryUsage functions.

Usage:

```
    'use strict'

    setTimeout(() => {
        const uptime = process.uptime()
        const { user, system } = process.cpuUsage()
        const stats = [process.memoryUsage()]

        console.log(uptime, user, system, (user + system)/1000000)
    }, 1000)


```

12.4 System Info

The os module can be used to get information about the Operating System.

```
  'use strict'
  const os = require('os')

  console.log('Hostname', os.hostname())
  console.log('Home dir', os.homedir())
  console.log('Temp dir', os.tmpdir())
```

There are two ways to identify the Operating System with the os module: the os.platform function which returns the same as process.platform property, and the os.type function which on non-Windows systems uses the uname command and on Windows it uses the ver command.

```
    'use strict'
    const os = require('os')

    console.log('platform', os.platform())
    console.log('type', os.type())
```

12.5 System Stats

```
    'use strict'
    const os = require('os')

    setInterval(() => {
        console.log('system uptime', os.uptime())
        console.log('freemem', os.freemem())
        console.log('totalmem', os.totalmem())
        console.log()
    }, 1000)
```

13 Child processes

13.1 Creating Child Processes

13.1.1 execFile and execFileSync Methods

The execFile and execFileSync methods are variations of the exec and execSync methods. Rather than defaulting to executing a provided command in a shell, it attempts to execute the provided path to a binary executable directly.

13.1.2 fork

The fork method is a specialization of the spawn method. By default, it will spawn a new Node process of the currently executing JavaScript file (although a different JavaScript file to execute can be supplied). It also sets up Interprocess Communication (IPC) with the subprocess by default.

13.1.3 exec and execSync Methods

The child_process.execSync method is the simplest way to execute a command.

```
    'use strict'

    const { execSync } = require('child_process')
    const output = execSync(
    `node -e "console.log('subprocess stdio output')"`
    )
    console.log(output.toString())
```

The execSync method returns a buffer containing the output (from STDOUT) of the command.

If we do want to execute the node binary as a child process, it's best to refer to the full path of the node binary of the currently executing Node process. This can be found with process.execPath. Using process.execPath ensures that no matter what, the subprocess will be executing the same version of Node.

The following is the same example from earlier, but using process.execPath in place of just 'node':

```
    'use strict'
    const { execSync } = require('child_process')
    const output = execSync(
    `${process.execPath} -e "console.error('subprocess stdio output')"`
    )
    console.log(output.toString())
```

The exec method takes a shell command as a string and executes it the same way as execSync. Unlike execSync the asynchronous exec function splits the STDOUT and STDERR output by passing them as separate arguments to the callback function:

```
    'use strict'
    const { exec } = require('child_process')

    exec(
    `"${process.execPath}" -e "console.log('A');console.error('B')"`,
    (err, stdout, stderr) => {
        console.log('err', err)
        console.log('subprocess stdout: ', stdout.toString())
        console.log('subprocess stderr: ', stderr.toString())
    }
    )
```

13.1.4 spawn and spawnSync Methods

While exec and execSync take a full shell command, spawn takes the executable path as the first argument and then an array of flags that should be passed to the command as second argument:

```
    'use strict'
    const { spawnSync } = require('child_process')

    const result = spawnSync(
```

```
    //  executable path
    process.execPath,
    // array of flags to be passed to spawned process
    ['-e', `console.log('subprocess stdio output')`]
    )
    // While the execSync function returns a buffer
    // containing the child process output,
    // the spawnSync function returns an object containing
    // information about the process that was spawned.
    console.log(result)
```

Unlike execSync, the spawnSync method does not need to be wrapped in a try/catch. If a spawnSync process exits with a non-zero exit code, it does not throw:

```
    'use strict'
    const { spawnSync } = require('child_process')
    const result = spawnSync(process.execPath, [`-e`, `process.exit(1)`])
    console.log(result)
```

Just as there are differences between execSync and spawnSync there are differences between exec and spawn.

While exec accepts an optional callback, spawn does not. Both exec and spawn return a ChildProcess instance however, which has stdin, stdout and stderr streams and inherits from EventEmitter allowing for exit code to be obtained after a close event is emitted:

```
    'use strict'
    const { spawn } = require('child_process')

    const sp = spawn(
    process.execPath,
    [`-e`, `console.log('subprocess stdio output')`]
    )

    console.log('pid is', sp.pid)

    // get the STDOUT of the child process
    sp.stdout.pipe(process.stdout)

    sp.on('close', (status) => {
        console.log('exit status was', status)
    })
```

There is one highly important differentiator between spawn and the other three methods we've been exploring (namely exec, execSync and spawnSync): the spawn method is the only method of the four that *doesn't buffer child process output.*

Even though the exec method has stdout and stderr streams, they will stop streaming once the subprocess output has reached 1 mebibyte (or 1024 * 1024 bytes). This can be configured with a maxBuffer option, but no matter what, the other three methods have an upper limit on the amount of output a child process can generate before it is truncated.

Since the spawn method does not buffer at all, it will continue to stream output for the entire lifetime of the subprocess, no matter how much output it generates. Therefore, for long running child processes it's recommended to use the spawn method.

13.2 Process Configuration

An options object can be passed as a third argument in the case of spawn and spawnSync or the second argument in the case of exec and execSync.

We'll explore two options that can be passed which control the environment of the child process: cwd and env.

We'll use spawn for our example but these options are universally the same for all the child creation methods.

By default, the child process inherits the environment variables of the parent process:

```
'use strict'
const { spawn } = require('child_process')

process.env.A_VAR_WE = 'JUST SET'
const sp = spawn(process.execPath, ['-p', 'process.env'])
sp.stdout.pipe(process.stdout)
```

If we pass an options object with an env property the parent environment variables will be overwritten:

```
'use strict'

const { spawn } = require('child_process')

process.env.A_VAR_WE = 'JUST SET'

const sp = spawn(process.execPath, ['-p', 'process.env'], {
    env: {SUBPROCESS_SPECIFIC: 'ENV VAR'}
})

sp.stdout.pipe(process.stdout)
```

Another option that can be set when creating a child process is the cwd option:

```
'use strict'
const { IS_CHILD } = process.env

if (IS_CHILD) {
    console.log('Subprocess cwd:', process.cwd())
    console.log('env', process.env)
} else {
    const { parse } = require('path')
    const { root } = parse(process.cwd())
    const { spawn } = require('child_process')
    const sp = spawn(process.execPath, [__filename], {
        cwd: root,
        env: {IS_CHILD: '1'}
    })

    sp.stdout.pipe(process.stdout)
}
```

13.3 Child STDIO

So far we've covered that the asynchronous child creation methods (exec and spawn) return a ChildProcess instance which has stdin, stdout and stderr streams representing the I/O of the subprocess.

This is the default behavior, but it can be altered.

Let's start with an example with the default behavior:

```
'use strict'
const { spawn } = require('child_process')
const sp = spawn(
process.execPath,
[
'-e',
`console.error('err output'); process.stdin.pipe(process.stdout)`
],
// default
{ stdio: ['pipe', 'pipe', 'pipe'] }
)

```

```
    sp.stdout.pipe(process.stdout)
    sp.stderr.pipe(process.stdout)
    sp.stdin.write('this input will become output\n')
    // ends the input stream, allowing the child process to exit
    // which in turn allows the parent process to exit.
    sp.stdin.end()
```

If we're piping the subprocess STDOUT to the parent process STDOUT without transforming the data in any way, we can instead set the second element of the stdio array to 'inherit'. This will cause the child process to inherit the STDOUT of the parent:

We've changed the stdio[1] element from 'pipe' to 'inherit' and . This will result in the exact same output:

```
    'use strict'
    const { spawn } = require('child_process')
    const sp = spawn(
    process.execPath,
    [
    '-e',
    `console.error('err output'); process.stdin.pipe(process.stdout)`
    ],
    { stdio: ['pipe', 'inherit', 'pipe'] }
    )

    sp.stderr.pipe(process.stdout)
    // removed the sp.stdout.pipe(process.stdout) line
    // (in fact sp.stdout would now be null)
    sp.stdin.write('this input will become output\n')
    sp.stdin.end()
```

The stdio option can also be passed a stream directly. In our example, we're still piping the child process STDERR to the parent process STDOUT. Since process.stdout is a stream, we can set stdio[2] to process.stdout to achieve the same effect:

Now both sp.stdout and sp.stderr will be null because neither of them are configured to 'pipe' in the stdio option.

```
    'use strict'
    const { spawn } = require('child_process')
    const sp = spawn(
    process.execPath,
    [
    '-e',
    `console.error('err output'); process.stdin.pipe(process.stdout)`
```

```
    ],
    // now sp.stderr will be null as well
    { stdio: ['pipe', 'inherit', process.stdout] }
    )

    sp.stdin.write('this input will become output\n')
    sp.stdin.end()
```

To send input to a child process created with spawn or exec we can call the write method of the return ChildProcess instance (sp.stdin.write('...
n')). For the spawnSync and execSync functions an input option be used to achieve the same:

```
    'use strict'
    const { spawnSync } = require('child_process')

    spawnSync(
    process.execPath,
    [
    '-e',
    `console.error('err output'); process.stdin.pipe(process.stdout)`
    ],
    {
        input: 'this input will become output\n',
        stdio: ['pipe', 'inherit', 'ignore']
    }
    )
```

For the input option to work for spawnSync and execSync the stdio[0] option has to be pipe, otherwise the input option is ignored.

14 Writing Unit Tests

14.1 Assertions

The core assert module exports a function that will throw an AssertionError when the value passed to it is falsy (meaning that the value can be coerced to false with !!val):

The core assert module has the following assertion methods:

```
    assert.ok(val) - the same as assert(val)

    assert.equal(val1, val2) - val1 == val2
    assert.notEqual(val1, val2) -val1 != val2
```

```
5
6     // also checks type
7     assert.strictEqual(val1, val2) - val1 === val2
8     assert.notStrictEqual(val1, val2) - val1 !== val2
9
10    // all values in an object
11    assert.deepEqual(obj1, obj2)
12    assert.notDeepEqual(obj1, obj2)
13
14    // al values in an object, also hecking type
15    assert.deepStrictEqual(obj1, obj2)
16    assert.notDeepStrictEqual(obj1, obj2)
17
18    assert.throws(function)
19    assert.doesNotThrow(function)
20
21    assert.rejects(promise||async function) -assert promise rjects
22    assert.doesNotReject(promise||async function) -assert promise resolves
23
24    assert.ifError(err) - check that an error object is falsy
25
26    assert.match(string, regex)
27    assert.doesNotMatch(string, regex)
28
29    assert.fail() -force an AssertionError to be thrown
```

The assert module also exposes a strict object where namespaces for non-strict methods are strict, so the above code could also be written as:

```
1     const assert = require('assert')
2     const add = require('./get-add-from-somewhere.js')
3     assert.strict.equal(add(2, 2), 4)
```

It's worth noting that assert.equal and other non-strict (i.e. coercive) assertion methods are deprecated, which means they may one day be removed from Node core. Therefore if using the Node core assert module, best practice would be always to use assert.strict rather than assert, or at least always use the strict methods (e.g. assert.strictEqual).

The error handling assertions (throws, ifError, rejects) are useful for asserting that error situations occur for synchronous, callback-based and promise-based APIs.

Let's start with an error case from an API that is synchronous:

```
1     const assert = require('assert')
2     const add = (a, b) => {
```

```
        if (typeof a !== 'number' || typeof b !== 'number') {
            throw Error('inputs must be numbers')
        }
        return a + b
    }
    // need to pass a function
    assert.throws(() => add('5', '5'), Error('inputs must be numbers'))
    assert.doesNotThrow(() => add(5, 5))
```

Notice that the invocation of add is wrapped inside another function. This is because the assert.throws and assert.doesNotThrow methods have to be , which they can then wrap and call to see if a throw occurs or not. When executed the above code will pass, which is to say, no output will occur and the process will exit.

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

15 Appendix

15.1 JavaScript Function Definitions

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

15.1.1 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.

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

15.1.2 Function expression (unnamed)

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

```

    // create a method on an object
    { sum: function() {...},
      //
    }

    // use as a callback
    array.map(function(...) {...})

```

15.1.3 Function expression (named)

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

```

    const isTruthy = function(value) {
        return !!value;
    }
    // or
    let count = function(...) {...}

```

15.1.4 Shorthand method definition

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

```
const collection = {
    items: [],
    add(...items) {
        this.items.push(...items);
    },
    get(index) {
        return this.items[index];
    }
};

collection.add('C', 'Java', 'PHP');

```

15.1.5 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).

```
const absValue = (number) => {
    if (number < 0) {
        return -number;
    }
    return number;
}

```

15.1.6 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 *.

```
const obj = {
    *indexGenerator() {
```

```
            var index = 0;
            while(true) {
                yield index++;
            }
        }
    }

```

```
    he events module exports an EventEmitter constructor:

    const { EventEmitter } = require('events')

    In modern node the events module is the EventEmitter constructor as well:

    const EventEmitter = require('events')

    Both forms are fine for contemporary Node.js usage.

    To create a new event emitter, call the constructor with new:

    const myEmitter = new EventEmitter()

    A more typical pattern of usage with EventEmitter, however, is to inherit from it:

    class MyEmitter extends EventEmitter {
        constructor (opts = {}) {
            super(opts)
            this.name = opts.name
        }
    }

```