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:

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

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

I 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:

```
async function run () {
    const big = readFile(bigFile)
    const medium = readFile(mediumFile)
    const small = readFile(smallFile)

    big.then(print)
    medium.then(print)
    small.then(print)

    await small
    await medium
    await big
```

```
}

run().catch(console.error)

```

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

```

```

```

```

```

```