1 The Node Binary

1.1 Common Command Line Arguments

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

1.2 Module availability

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

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

1.3 Preloading files

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

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

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

1.4 Stack trace limit

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

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

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

2 Debugging and Diagnostics

Start node in debugging mode:

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

3 Core JavaScript Concepts

3.1 Types

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

3.1.1 Primitive Types

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

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

3.1.2 Object

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

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

3.1.3 Functions

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

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

Lambda functions do not have a prototype.

3.2 Prototypal Inheritance

3.2.1 Prototypal inheritance - functional

Create a prototype chain:

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

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

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:

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

Define a function to set up the inheritance chain:

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

Define how to obtain a child object:

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

Create a child object():

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

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

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

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

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

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

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

3.2.3 Prototypal inheritance - class-based

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

Usage:

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

3.3 Inheritance provided by closures

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

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

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

4 Packages and Dependencies

4.1 Specifying a SemVer range

- Prefix the version with a caret (ˆ) to include everything that does not increment the first non-zero portion of semver. Example: ˆ8.14.1 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: ˆ2 <2.2 || > 2.3.

5 Module System

5.1 Detecting Main Module in CJS

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

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

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

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

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

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

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

5.2 Converting a Local CJS File to a Local ESM File

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

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

require() cannot be used with ESM modules.

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

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

5.2.1 Aside: Static and Dynamic Imports

Assume we have a file utils.mjs

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

This is a static import:

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

Whereas this is a dynamic import:

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

5.3 Converting a CJS Package to an ESM Package

5.3.1 Specify module type

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

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

5.3.2 Exports in ESM

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.3.3 Importing modules

With static imports, different import possibilities exist:

- Implicitly import a module's default export

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

- Import a specific named export from a module

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

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

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

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

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

```
1
```

```
1
```

```
1
```