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

```
1  // The null primitive is typically used to describe
2  // the absence of an object...
3  // Null
4  null
5
6  // Undefined
7  // ... whereas undefined is the absence
8  // of a defined value.
9  // Any variable initialized without a value will be undefined.
10 // Any expression that attempts access of a non-existent
11 property on an object will result in undefined.
12 // A function without a return statement will return undefined.
13 // undefined
```

```
14
15        // Number
16        // The Number type is double-precision floating-point format.
17        // It allows both integers and decimals but
18        // has an integer range of $-2^53-1$ to $2^53-1$.
19        1, 1.5, -1e4, NaN
20
21        // BigInt
22        // The BigInt type has no upper/lower limit on integers.
23        1n, 9007199254740993n
24
25        // String
26        'str', "str", `str ${var}`
27
28        // Boolean
29        true, false
30
31        // Symbol
32        // Symbols can be used as unique identifier keys in objects.
33        //The Symbol.for method creates/gets a global symbol.
34        Symbol('description'), Symbol.for('namespace')...
35
36
```

3.1.2 Object

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

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

3.1.3 Functions

```
1         // this refers to the object on which the function was called,
2         // not the object the function was assigned to
3         const obj = { id: 999, fn: function () { console.log(this.id) } }
4         const obj2 = { id: 2, fn: obj.fn }
5         obj2.fn() // prints 2
6         obj.fn() // prints 999
7
```

```
        // 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') }
    }
```

```
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: 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')
```

```
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: awooooooooo"
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```

```
1
```