Udemy NodeJs 1-54

Udemy:
https://www.udemy.com/understand-nodejs/learn/v4/overview

Command Line Commands
https://www.computerhope.com/issues/chusedos.html
https://www.lifewire.com/list-of-command-prompt-commands-4092302

NodeJs Command Line
https://nodejs.org/api/cli.html

Node,js Native Modules:
https://nodejs.org/api/

Section 2 V8 javascript engine
Lesson 6 Processors Machine Language and c++
Lesson 7 Javasript Engines and EcmaScript specification
Lesson 8 V8 Under the Hood
Lesson 9 Adding Features to Javascript
Section 3 Node Core
Lesson 10 Servers and Clients
Section 4 Modules, Export require
Lesson 16 Modules
Lesson 18 Let's Build a Module
Lesson 25 More on require
Lesson 26 Module Patterns
Lesson 27 Exports default module.exports
Lesson 28 Require Native (Core) Modules
Lesson 29 Module ES6 Import
Section 5 Events
Lesson 31 Events
Lesson 32 Node Event Emitter
Lesson 36 Inheriting Event Emitter
Lesson 37 Node ES6, Template Literals
Lesson 39 Inhteriting from the Event emitter part 2
Lesson 40 ES6 Classes
Lesson 41 Inheriting from the Event emitter part 3
Section 6 Async
Lesson 44 libuv, The Event Loop, and Non-Blocking Asynchronous Execution
Lesson 55 Streams and Buffers
Lesson 46 Binary Data, Character Sets, and Encodings
Lesson 47 Buffer
Lesson 48 ES6 Typed Arrays
Lesson 49 Callbacks
Lesson 50 Files
Lesson 51 Streams
Lesson 52 Pipes Theory
Lesson 53 Pipes Example

Lesson 6 Processors Machine Language and c++

Microprocessor: Very small machine, which do a job, with instructions. They **speak machine language**.

```
000018A45438100      0  55              push rbp
0000018A45438101     1  4889e5          REX.W movq rbp,rsp
0000018A45438104     4  56              push rsi
0000018A45438105     5  57              push rdi
0000018A45438106     6  41ff75a8        push [r13-0x58]
0000018A4543810A    10  56              push rsi
0000018A4543810B    11  49baf9552c7e8f010000 REX.W movq r10,
0000018F7E2C55F9    ;;  object: 0000018F7E2C55F9
0000018A45438115    21  4152            push r10
0000018A45438117    23  6a00            push 0x0
0000018A45438119    25  b803000000      movl rax,000000000000
```

Node is written in C+++.
V8 is written in c++.
We are writing in javascript which willl be converted into c++, which will be converted to machine code.

Lesson 7 Javasript Engines and EcmaScript specification

EcmaScript: Is the standard javascript is based on. (Core Standard)

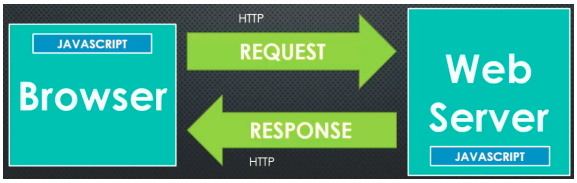
Lesson 8 V8 Under the Hood

Google's open source javascript engine, use ECMAScript. It is written in C++.

Lesson 9 Adding Features to Javascript

Since V8 is opensource, you can donwload it, and with c++ you can make additional features.

Lesson 10 Servers and Clients



Lesson 16 Modules

Module: Resuable block of code, which does not impact with other code acctidentaly

CommonJs: a set of standard based on which the modules should be written

Lesson 18 Let's Build a Module

1. Write the funciton
2. Export module
3. require

1,2

```
//We write the code, storing in a variable, memory space
var greet = function() {
    console.log('Hello!');
};

//We want to make the attached object available to use outside the module
module.exports = greet;
```

3

```
var greet = require('./greet');
greet();
```

Lesson 25 More on require

1. You can import whole folders, but when you do this, it is looking for **index.js**

```
var greet = require('./greet');
```

2. export the seperate js files to the same variable name "**greet**"

spanish.js

```
var greetings = require('./greetings.json');
```

```
var greet = function() {
    console.log(greetings.es);
}

module.exports = greet;
```

english.js

```
//Inject required modules into the functions
var greetings = require('./greetings.json');

var greet = function() {
    console.log(greetings.en);
}

module.exports = greet;
```

Create a seperate object, which contains the exported/required js files
index.js

```
var english = require('./english');
var spanish = require('./spanish');

// Export multiple modules
module.exports = {
    english: english,
    spanish: spanish
};
```

After they have been connected, you can actually just import everything with 1 require.
app.js

```
var greet = require('./greet');

greet.english();
greet.spanish();
```

Lesson 26 Module Patterns

1. Exporting directly the function

```
//We are exporting the function pointing to the export property of the module object.
module.exports = function() {
    console.log('Hello world');
};
```

```
//We require the variable,pointing to te function
var greet = require('./greet1');
greet();
```

2. Adding a property(function) to the module.export object

```
module.exports.greet = function() {
    console.log('Hello world!');
};
```

```
// We say explicitly, that we ar looking for the exported property of the module.export object
var greet2 = require('./greet2').greet;
greet2();
```

3. Using function constructor and "new" keyword to rerun a new created object from the module

```
function Greetr() {
    this.greeting = 'Hello world!!';
    this.greet = function() {
        console.log(this.greeting);
```

```
    }
}
module.exports = new Greetr();
```

Since node.js module, caches the exported modules, this way, we always referencing the same object

```
//We are importing and creating a new object, by importing the module, which returning a new object with the keyword of "new"
var greet3 = require('./greet3');
greet3.greet();
greet3.greeting = 'Changed hello world!';

// Require stores, caches the  already exported modules,
// So by requiring it again, it will just create a reference to the already created object
var greet3b = require('./greet3');
greet3b.greet();
```

4. Exporting only the function constructor

```
function Greetr() {
    this.greeting = 'Hello world!!!';
    this.greet = function() {
        console.log(this.greeting);
    }
}
module.exports = Greetr;
```

If we want to have different objects, we just have to use the "new" keyword for the required modules

```
//If we want to have different objects, we can require the module, and use the "new" operator
//So the exported stuff is the function constructor
var Greet4 = require('./greet4');
```

```
var grtr = new Greet4();

grtr.greet();
```

5.Revealing, exposing only a certain function, part of the module

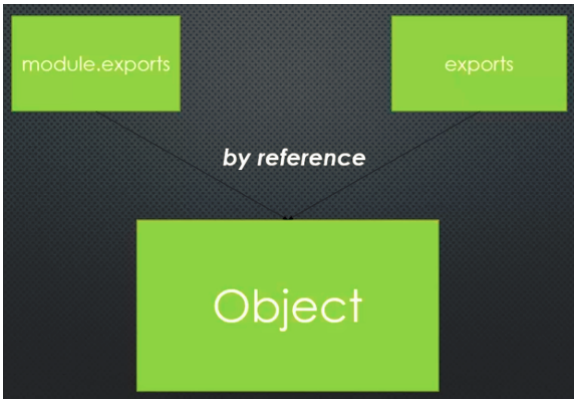
```
 //Revealing.We are only exposing the function, reveal to outside of the module


var greeting = 'Hello world!!!!';


function greet() {
    console.log(greeting);
}
//When this is executed, it will still have access to greetin variable, but outside of the module we have no reference
module.exports = {
    greet: greet
}
```

So we are referencing the exported, property of the object.

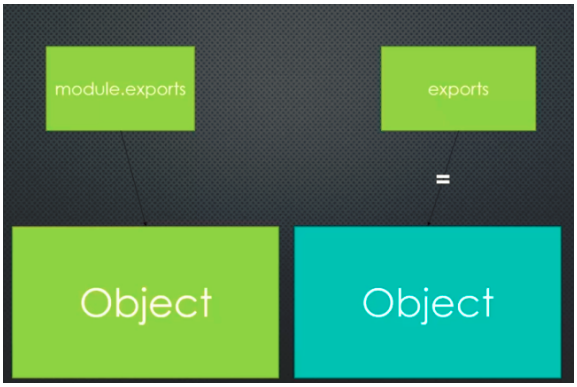
```
var greet5 = require('./greet5').greet;

greet5();
```

Lesson 27 Exports default module.exports

If we are make a object to equal other object, instead of overwriting it, we share the reference, so two variables ar pointing at the same space in memory



But if we equal a variable to a value, to a new function object, it's original reference will be overwritten, the connection will be broken.



1. Overwriting the orignal exports object, greet.js

```
//Since we are overwriting the exports object to point to a new function object, this will not pointing to the exports object
exports = function() {
    console.log('Hello');
}

console.log(exports);           //overwritten object
console.log(module.exports);    //Original object
```

2. Mutate the orignal object, add a new property to the ojbect, greet2.js

```
//We are mutating the object, by adding a new property to the export object
exports.greet = function() {
    console.log('Hello');
};

console.log(exports);       // {}          //They are poiting to the same function
console.log(module.exports);// {}
```

app.js

```
var greet = require('./greet');         //not working, since we los the original
reference
var greet2 = require('./greet2');       //Working since, we kept the original ojbect
reference
greet2.greet();
```

Lesson 28 Require Native (Core) Modules

Native modules:
Codes, which are shipped with **nodejs**
https://nodejs.org/api/

We are not referencing the folder, it will look at the utility core folder

```
var util = require('util');


var name = 'Tony';
var greeting = util.format('Hello, %s', name);
util.log(greeting);
```

Lesson 29 Module ES6 Import

require and import is basically the same

Lesson 31 Events

Event: Sth that has happened in our app that we can respond to.

System events: (from C++ core, libuv)
Finished reading file, data has been recieved etc.

Custom events: (Javascript)
Create custom events, (**Event Emitter**)

Lesson 32 Node Event Emitter

So basically, we are **creating an object** with **function constructor** and **"new"** keyword, which contains properties, with an array of code to be execute, when we are calling a certain propery of the object.

emitter.js

```
//Function constructor, which is basically an events object
function Emitter() {
    this.events = {};
}
// variable = conditional || false conditional
//Adding a function to the prototype, which put the type in the type array, and a code to be executed (hash:Table)
Emitter.prototype.on = function(type, listener) {
    this.events[type] = this.events[type] || [];      // if the poperty exist, than that is all great, if not, creat an empty array
    this.events[type].push(listener);                 // We put an code into the array, to be executed when that event happened
}

//TO say that sth happened, we get the type of the event, and we are looping through the array of the selected event
Emitter.prototype.emit = function(type) {
    if (this.events[type]) {
        this.events[type].forEach(function(listener) {
            listener();
        });
    }
}
//Creating a simple module
module.exports = Emitter;
```

app.js

```
var Emitter = require('./emitter');


var emtr = new Emitter();


//put properties to the event object
emtr.on('greet', function() {
    console.log('Somewhere, someone said hello.');
});


emtr.on('greet', function() {
    console.log('A greeting occurred!');
});


//Execute the created array of listeners for the event.
console.log('Hello!');
emtr.emit('greet');
```

MagicString: A string that has some special meaning in our app.
Do not rely in the strings!

Instead of relying string calling, we wrap it into an imported encapsulated variable, so it can be debugged more easily.
(Lesson 32) Magic String

```
emtr.on('greet', function() {
    console.log('Somewhere, someone said hello.');
});
```

Create a seperated encapsulated object
config.js

```
module.exports = {
```

```
    events: {
        GREET: 'greet'
    }
}
```

app.js

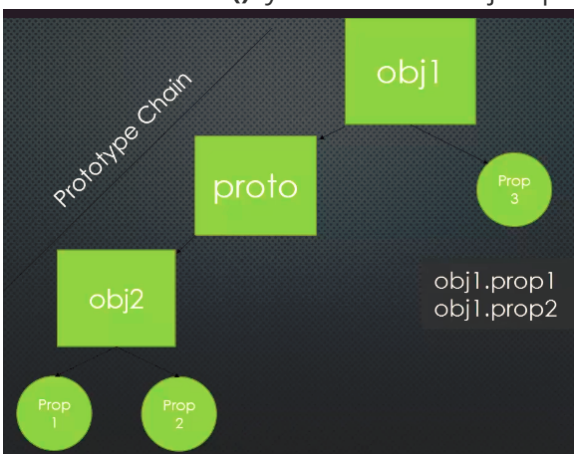
```
var eventConfig = require('./config').events;

emtr.on(eventConfig.GREET, function() {
    console.log('Somewhere, someone said hello.');
});
```

Lesson 36 Inheriting Event Emitter

// It will let you to share the EventEmitter's propert with the Greetr functional constructor

```
util.inherits(Greetr, EventEmitter);
```

with **util.inherits()** you can share object properties from top to down.




```
var EventEmitter = require('events');
var util = require('util');

//Functional constructor
function Greetr() {
```

```
    this.greeting = 'Hello world!';
}

// It will let you to share the EventEmitter's propert with the Greetr functional
constructor
// It has to be done, prior assigning any function to it!
util.inherits(Greetr, EventEmitter);

//Add method to the prototpye
Greetr.prototype.greet = function(data) {
    console.log(this.greeting + ': ' + data);
    this.emit('greet', data);                          //Calling the emitted data
}
//Creating a Greetr Object
var greeter1 = new Greetr();
//Adding listener function to the "greet" event
greeter1.on('greet', function(data) {
    console.log('Someone greeted!: ' + data);
});

greeter1.greet('Tony');
```

Lesson 37 Node ES6, Template Literals

Template Literals

```
var name = 'John Doe';

var greet = 'Hello ' + name;
var greet2 = `Hello ${ name }`;
```

You have to infrom the browser for backward compatibility.

Create this file in the folder where the ES6 js file is located.

jsconfig.json

```
{
    "compilerOptions": {
        "target": "ES6"
    }
}
```

Lesson 39 Inhteriting from the Event emitter part 2

You can create a base object, and basically extend it's propeties and methods, to newly created objects and connection them with the prototypal chain.

```
var util = require('util');                         //Imported for the inherits function

//Functional constructor for a new pderson object
function Person() {
    this.firstname = 'John';
    this.lastname = 'Doe';
}


//Adding a basic option to the greet function
Person.prototype.greet = function() {
    console.log('Hello ' + this.firstname + ' ' + this.lastname);
}
//Creating the Policement functional constructor
function Policeman() {
    console.log("Policeman Constructor");
    console.log(this);          //It is an empty object, created by this functional
constructor
    Person.call(this);          //Passing the object reference to the borrowed object's
(from we will have all of the properties and methods)
    this.badgenumber = '1234';
```

```
}

//We establish intheriance, connection between the two objects

util.inherits(Policeman, Person);

var officer = new Policeman();

officer.greet();

console.log(officer);

console.log(officer.badgenumber);
```

Lesson 40 ES6 Classes

New way to create objects, syntactically.

You can use **class**, **extends class** to create object, with prototypal inheritance more easily, but everything is the same under the hood
'use strict';

```
//We have the base object, with the constructor function, with a greet metho

class Person {

    constructor(firstname, lastname) {

        this.firstname = firstname;

        this.lastname = lastname;

    }


    greet() {

        console.log('Hello, ' + this.firstname + ' ' + this.lastname);

    }

}


//We can just simply use it's function

var john = new Person('John', 'Doe');

john.greet();


var jane = new Person('Jane', 'Doe');
```

```
jane.greet();


console.log(john.__proto__);
console.log(jane.__proto__);
console.log(john.__proto__ === jane.__proto__);         //The prototype of the created
classes are the same



class Policeman extends Person {
  constructor(firstname, lastname) {            //This is for passing in properties
    super(firstname, lastname);                 //Borrowing the extended object's
properties, methods
    this.badgnumber = '1234';
  }

  getBadge() {                                  //Adding custom method to the new
object only
    console.log(this.badgnumber);

  }
}

let officer = new Policeman("Police","John");
console.log(john);
console.log(officer);
officer.getBadge();
officer.greet();
```

Lesson 41 Inheriting from the Event emitter part 3

Creating the prototypal intheritance using ES6 class in the Greetr Object, which we import
In itself the greet function, we use the emit, to execute the code for the hardcoded event (Magic String)
greeter.js

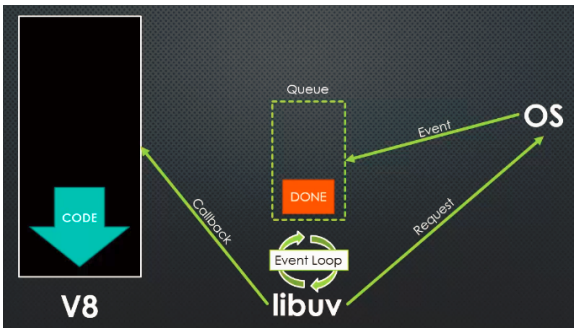
```
//So here we are importing the EventEmitter object,
// We are extending it, making the  EventEmitter object a prototype of the new Greetr object
//Which has all of the prototype properties using super()
//+ we are adding a custom method
'use strict';
var EventEmitter = require('events');
module.exports = class Greetr extends EventEmitter {
    constructor() {
        super();
        this.greeting = 'Hello world!';
    }
    greet(data) {
        console.log(`${ this.greeting }: ${ data }`);
        this.emit('greet', data);
    }
}
```

App.js

```
//Creating a Greetr object, from the extended EventEmitter object class, than assign an event emitter to it
//And i the  Greetr object's greet() function, we are emitting the event, so it will be triggered.
'use strict';
var Greetr = require('./greetr');
var greeter1 = new Greetr();

greeter1.on('greet', function(data) {
    console.log('Someone greeted!: ' + data);
});

greeter1.greet('Tony');
```

Lesson 44 libuv, The Event Loop, and Non-Blocking Asynchronous Execution

libuv (c++) is managing the custom events EventEmmitter
It will make a request to the **O**perating**S**ystem, which will put the finished event to the **Queue** of libuv, which will check it regulary with the **event loop**, then notify the **v8 engine** that the request has been finished (**callback**) which will tell the v8 engine to **execute some code.**



Lesson 55 Streams and Buffers

Buffer: A temporary holding spot for data.

Stream: A sequence of data made available overtime, which will be combined into a whole.

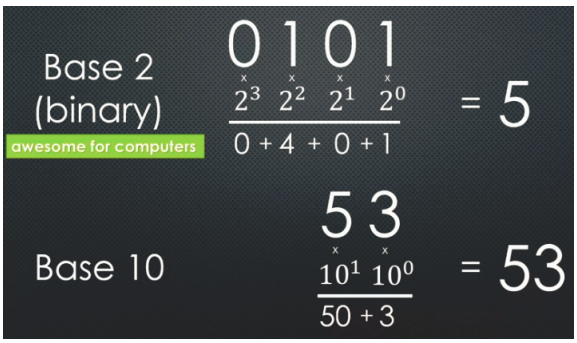
Stream is gathering data into the buffer

When buffer is full

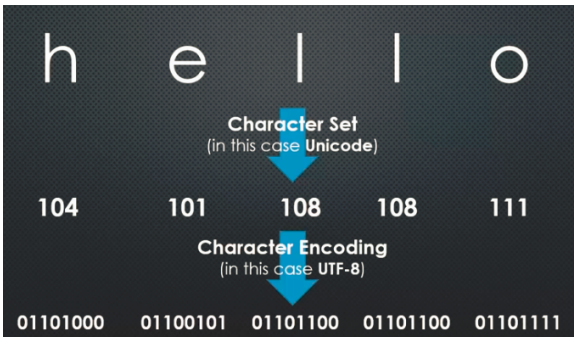
Will move it to the processors and start fetching new chunck of data in the stream

Lesson 46 Binary Data, Character Sets, and Encodings

Binary Data: Data stored in binary, in sets of 0 and 1. (bit-binary data)



Character set: A representation of characters as numbers. (**Unicode, ASCII**)
UTF_8 (we can display characters in 8 bits.)
Encoding: How many bits are we using to store the character set number.



Lesson 47 Buffer

```
//Buffer is global!
// buffer(String value, encoding)  -> take the string and encode it in utf8
var buf = new Buffer('Hello', 'utf8');
console.log(buf);                          //<Buffer 48 65 6c 6c 6f>, hexadecimal notation
console.log(buf.toString());               // Convert it back to characters
console.log(buf.toJSON());                 // We can convert it to JSOn to { type:
'Buffer', data: [ 72, 101, 108, 108, 111 ] }
console.log(buf[2]);                       // Get the character set number

buf.write('wo');                           //it behaves like an array, we can overwrite
it's characters like an array
console.log(buf.toString());               // From "He" + "llo" --> "wo" + "llo"
```

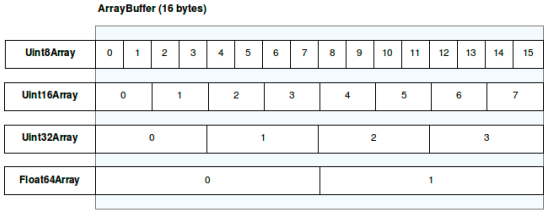
Lesson 48 ES6 Typed Arrays

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays

ArrayBuffer: It is just a squence of predefined number of bits (0 or 1)
Int32Array: It is generating a view, by storing small chunks of dat


```
//1. So we are declaring that we can store 8 byte(64bits),  as an array in our
temporary memory location called buffer.
//2. Typed Array,
var buffer = new ArrayBuffer(8);            // we can store 64 bits, (64 0 or 1)
var view = new Int32Array(buffer);          //We declare a array, which holds data
stored in 32 bits. So we can store big numbers, but only two.
view[0] = 5;
view[1] = 15;
console.log(view);
```

To achieve maximum flexibility and efficiency, JavaScript typed arrays split the implementation into **buffers** and **views**. A buffer (implemented by the `ArrayBuffer`object) is an object representing a chunk of data; it has no format to speak of and offers no mechanism for accessing its contents. In order to access the memory contained in a buffer, you need to use a view. A view provides a context — that is, a data type, starting offset, and the number of elements — that turns the data into a typed array.



Lesson 49 Callbacks

Since funcitons is javascript are first-class funcitons, we can pass them round as arguments and invoke them

```
function greet(callback) {
    console.log('Hello!');
    var data = {
        name: 'John Doe'
    };

    callback(data);
}

greet(function(data) {
    console.log('The callback was invoked!');
    console.log(data);
});

greet(function(data) {
    console.log('A different callback was invoked!');
    console.log(data.name);
});
```

Lesson 50 Files

```
var fs = require('fs');
//When our file is loaded, it will load it to the buffer, than to the view, the it will
be returned
//Synchronous call
var greet = fs.readFileSync(__dirname + '/greet.txt', 'utf8');
console.log(greet);
```

```
//Async call, to execute a call, when it will be finised reading.
var greet2 = fs.readFile(__dirname + '/greet.txt', 'utf8', function(err, data) {
    console.log(data);  //will read the binary data, and encode it in utf8
});

console.log('Done!');
```

error-first callback: Taken an error object as theri first parameter

Lesson 51 Streams

Send lot of chunk of data, via stream

1. Load data in the stream, without encoding it
2. Load the data and encode it ()
3. When we are loading the data and the buffer we can modify the data in the buffer

*** Since **Streams are EventEmitter.** There are built in emits and events, so every time when a buffer is ready, it will trigger an event, so it can be read, modified or anything.

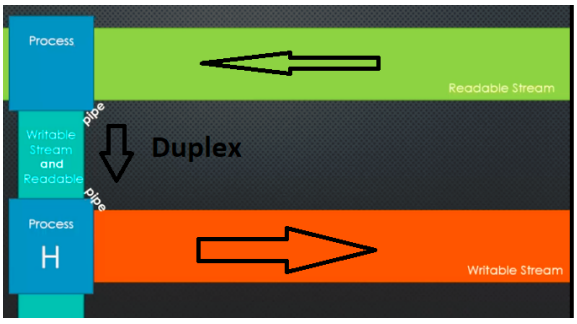
```
var fs = require('fs');

//***1. Without encoding, loads all of the buffer.***
var readable0 = fs.createReadStream(__dirname + '/greet.txt');

readable0.on('data', function(chunk) {
    console.log(chunk);
});

//***2.with encoding utf8, bufferSize is 16kb***
var readable = fs.createReadStream(__dirname + '/greet.txt', { encoding: 'utf8',
highWaterMark: 16 * 1024 });
```

```
// *** 3. create a writeStream, so we can write, modify the stream data, we are copying the data to an other txt
var writable = fs.createWriteStream(__dirname + '/greetcopy.txt');

//The stream will will a buffer with a content.
//If the buffer is smaller than the size of the file, you will get chunks of data.
//Since streaming, fs is an extended EventEmitter, you have the properties of it, you can emit.
readable.on('data', function(chunk) {
    console.log(chunk);
    console.log(chunk.lenght);
    writable.write(chunk);
});
```

Lesson 52 Pipes Theory

Pipe: Connection two stream, by writing to one stream what is being read from an other.

From a **readable** to a **writeable**.
So here, we are recieving a readable data, which will be redirected, by copy the chunk of data, to an other stream where you can read and write.
Then again, with pipe after processing you can redirect it to a writeable stream,



Lesson 53 Pipes Example

```
//Pipes are accessible methods on readable streams.
```

```
var fs = require('fs');
var zlib = require('zlib');                                    //Allows us to implement to a
gzip file, a particular algorytm to compress files


//Read a text from the stream, buffer
var readable = fs.createReadStream(__dirname + '/greet.txt');
//Write data in a stream and create txt
var writable = fs.createWriteStream(__dirname + '/greetcopy.txt');
//Writable stream, returns a zipped file
var compressed = fs.createWriteStream(__dirname + '/greet.txt.gz');
//compress the data with algorythmí
var gzip = zlib.createGzip();


//We read the data, then redirect it to the writable stream, which will makea copy of
txt
readable.pipe(writable);


//Since pipes are chainable, we can pipe different streams
//From the read data, we create a comporessed version, then since we are in a readable
& writeable stream, we create a writeable stream, which will create a zip file.
readable.pipe(gzip).pipe(compressed);
```

Lesson 54 Webserver Requirements