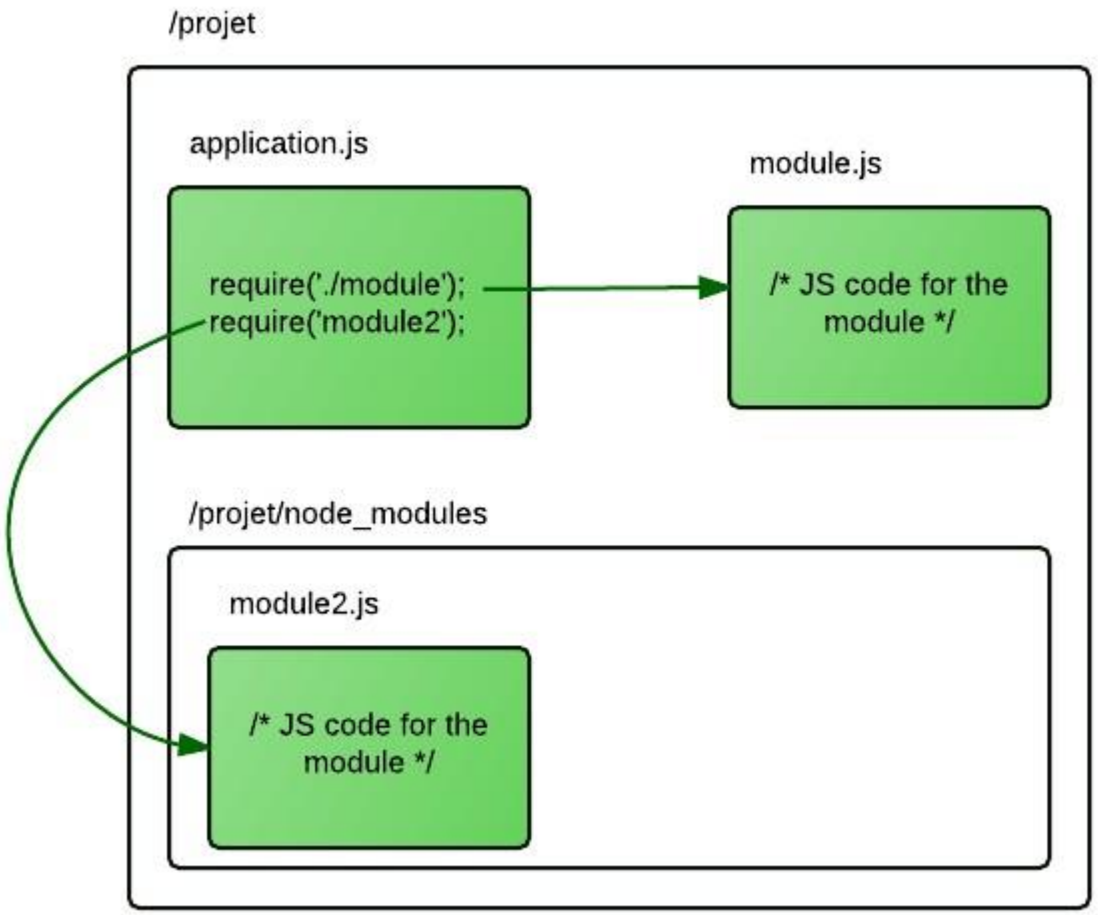
Modules

Modules in NodeJS

- Consider modules to be the same as JavaScript libraries.
- A set of functions you want to include in your application.

- Node implements **CommonJS** Modules specs.
  - CommonJS module are defined in normal `.js` files using `module.exports`
  - In Node.js, each file is treated as a separated module



https://openclassrooms.com

Type of Modules

- Built-in Modules
  - Node.js has a set of built-in modules which you can use without any further installation.
  - buffer, fs, http, path, etc.
  - Built-in Modules Reference
- 3rd party modules on www.npmjs.com
- Your own Modules
  - Simply create a normal JS file it will be a module *(without affecting the rest of other JS files and without messing with the global scope)*.

Include Modules – `require()` function

- The basic functionality of `require` is that it reads a JavaScript file, executes the file, and then proceeds to return the `module.exports` object.
  - `const path = require('path');`
  - `const config = require('./config');`
- Rules:
  - if the file doesn't start with "./" or "/", then it is either considered a **core module** (and the local Node path is checked), or a dependency in the local **node_modules** folder.
  - If the file starts with "./" it is considered a **relative file** to the file that called `require`.
  - If the file starts with "/", it is considered an **absolute path**.
  - If the filename passed to require is a directory, it will first look for `package.json` in the directory and load the file referenced in the main property. Otherwise, it will look for an `index.js`.
  - NOTE: you can omit "`.js`" and `require` will automatically append it if needed.

How `require('/path/to/file')` works

- Node goes through the following sequence of steps:
  1. Resolve: to find the absolute path of the file
  2. Load: to determine the type of the file content
  3. Wrap: to give the file its private scope
  4. Evaluate:This is what the VM does with the loaded code
  5. Cache: when we require this file again, don’t go over all the steps.

- **Note**: Node core modules return immediately (no resolve)

What’s the wrapper?

- `node -p "require('module').wrapper"`

1. Node will wrap your code into:

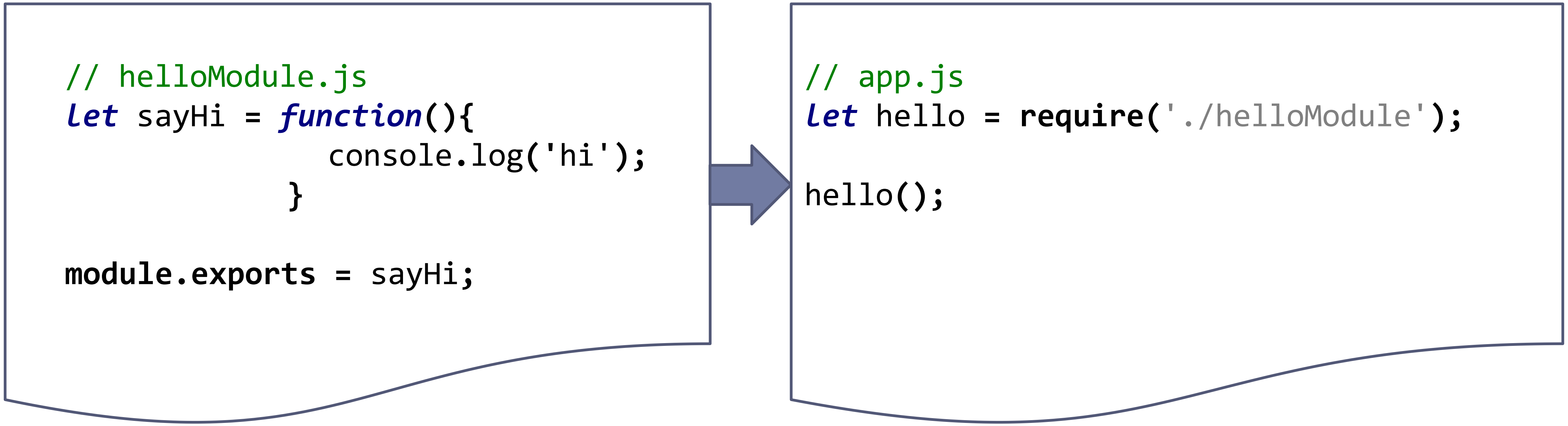
```
(function (exports, require, module, __filename, __dirname){
        exports = module.exports;
    // this is why can use exports and module objects.. etc in your code
    without any problem, because Node is going to initialize these and pass
    them as parameters through this wrapper function
});
```

2. Node will run the function using `.apply()`
3. Node will return the following:

```
return module.exports;
```

module.exports

- Think about this object (`module.exports`) as return statement.

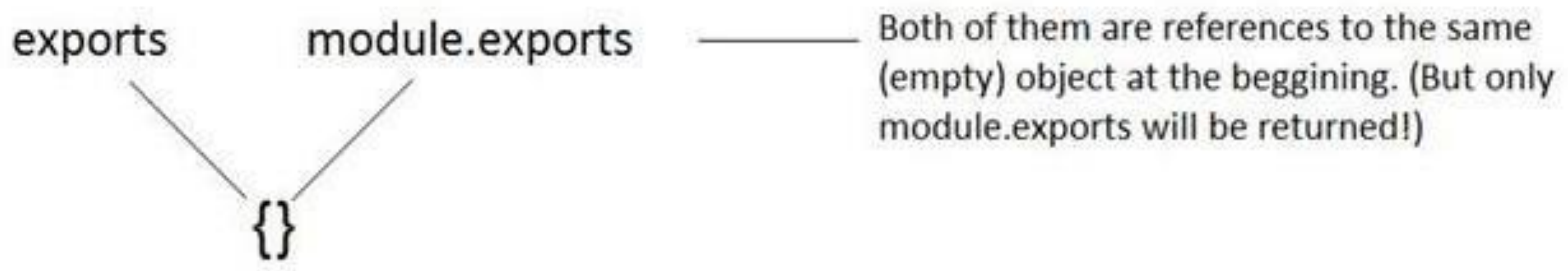


```javascript
// helloModule.js
let sayHi = function(){
               console.log('hi');
            }

module.exports = sayHi;
```

```javascript
// app.js
let hello = require('./helloModule');

hello();
```

`exports` vs `module.exports`

- `exports` object is a reference to the `module.exports`, that is shorter to type



Both of them are references to the same (empty) object at the beggining. (But only module.exports will be returned!)

- Be careful when using `exports`, a code like this will make it point to another object. At the end, `module.exports` will be returned.

```
exports = function doSmething() {
    console.log('blah blah');
}
```

doSomething() isn't exported.

Using Modules – Pattern 1

```
// Pattern1 - pattern1.js
module.exports = function () {
    console.log('Josh Edward');
};
```

```
// app.js
const getName = require('./pattern1');
getName(); // Josh Edward
```

Using Modules – Pattern 2

```
// Pattern2 - pattern2.js
module.exports.getName = function () {
    console.log('Josh Edward');
};

// OR
exports.getName = function () {
    console.log('Josh Edward');
};
```

```
// app.js
const getName = require('./pattern2').getName;
getName(); // Josh Edward

// OR
const person = require('./pattern2');
person.getName(); // Josh Edward
```

Using Modules – Pattern 3

```
// Pattern3 - pattern3.js
class Person {
    constructor(name) {
        this.name = name;
    }

    getName() {
        console.log(this.name
    }
}
module.exports = new Person('Josh Edward');
```

Warning: Not good practice

```
// Pattern3 - cached.js
const personObj2 = require('./pattern3');
 // cached
console.log('---inside cached.js ---');
personObj2.getName(); //Emma Smith
```

```
// app.js
const personObj = require('./pattern3');
personObj.getName(); // Josh Edward
personObj.name = 'Emma Smith';
personObj.getName(); //Emma Smith
const cachedObj = require('./pattern3');// cached in the same module
```

Using Modules – Pattern 4

```
// Pattern - pattern4.js
class Person {
    constructor(name = 'Josh Edward') {
        this.name = name;
    }

    getName() {
        console.log(this.name);
    }
}

module.exports = Person;
```

```
// app.js
const Person = require('./pattern4');
const personObj1 = new Person();
personObj1.getName() // Josh Edward
personObj1.name = 'Emma Smith';
personObj1.getName(); //Emma Smith

const Person2 = require('./pattern4');
const personObj2 = new Person2();
personObj2.getName(); // Josh Edward
```

Using Modules – Pattern 5

```
// Pattern5 - pattern5.js
const name = 'Josh Edward';
function getName() {
    console.log(name);
}
module.exports = {
    getName: getName // closure
}
```

```
// app.js
const getName = require('./pattern5').getName;
getName(); // Josh Edward
```

`path` module

- The `path` module provides a lot of very useful functionality to access and interact with the file system.

```
const path = require('path');

//Return the directory part of a path:
console.log(path.dirname('Buffer'));
console.log(path.dirname('File/example1.js')); // /test/something

//Joins two or more parts of a path:
const name = 'joe';
console.log(path.join('users', name, 'notes.txt'));
```

fs module

- The fs module provides a lot of very useful functionality to access and interact with the file system.

```
const fs = require('fs');
const path = require('path');
console.log(__dirname); // returns absolute path of current file
const greet = fs.readFileSync(path.join(__dirname, 'greet.txt'), 'utf8');
console.log(greet);

fs.readFile(path.join(__dirname, 'greet.txt'), 'utf8',
    function(err, data) { console.log(data); });
console.log('Done!');
// Hello // Done!// Hello
```

- Notice the Node Applications Design: any async function accepts a **callback as a last parameter** and the **callback function accepts error as a first parameter** (`null` will be passed if there is no error). Notice: `data` here is a buffer object. We can convert it with `toString` or add the encoding - `'utf8'`

Example Read/Write Files

```
const fs = require('fs');
const path = require('path');

// Reading from a file:
fs.readFile(path.join(__dirname, 'greet.txt'), { encoding: 'utf8' }, (err, data) =>{
    if (err) throw err;
    console.log(data);
});

// Writing to a file:
fs.writeFile('students.txt', 'Hello World!', (err) => {
    if (err) throw err;
    console.log('Done');
});
```

Stream

- Stream is a way to handle reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient way.

- Why streams?
  - Memory efficiency: you don't need to load large amounts of data in memory before you are able to process it
  - Time efficiency: it takes way less time to start processing data, since you can start processing as soon as you have it, rather than waiting till the whole data payload is available

- The Node.js stream module provides the foundation upon which all streaming APIs are built. All streams are instances of EventEmitter

Different types of streams

- Readable: a stream you can pipe from, but not pipe into (you can receive data, but not send data to it).When you push data into a readable stream, it is buffered, until a consumer starts to read the data. (`fs.createReadStream`)

- Writable: a stream you can pipe into, but not pipe from (you can send data, but not receive from it). (`fs.createWriteStream`)

Examples of Readable and Writable streams

Readable Streams

- HTTP responses, on the client
- HTTP requests, on the server
- fs read streams
- zlib streams
- crypto streams
- TCP sockets
- child process stdout and stderr
- process.stdin

Writable Streams

- HTTP requests, on the client
- HTTP responses, on the server
- fs write streams
- zlib streams
- crypto streams
- TCP sockets
- child process stdin
- process.stdout, process.stderr

Stream example

```
const fs = require('fs');
const path = require('path');

// Stream will read the file in chunks
// if file size is bigger than the chunk then it will read a chunk and emit a 'data' e
vent.
// Use encoding to convert data to String of hex
// Use highWaterMark to set the size of the chunk. Default is 64 kb

const readable = fs.createReadStream(path.join(__dirname, 'card.jpg'),
       { highWaterMark: 16 * 1024 });

const writable = fs.createWriteStream(path.join(__dirname, 'destinationFile.jpg'));

readable.on('data', function(chunk) {
    console.log(chunk.length);
    writable.write(chunk);
});
```

Pipes: src.pipe(dst);

- To connect two streams, Node provides a method called `pipe()` available on all readable streams. Pipes hide the complexity of listening to the stream events.

```
const fs = require('fs');
const path = require('path');

const readable = fs.createReadStream(path.join(__dirname, 'card.jpg'));
const writable = fs.createWriteStream(path.join(__dirname, 'destinationFile.jpg'));

readable.pipe(writable);

// note that pipe return the destination, this is why you can pipe it again to another
stream if the destination was readable in this case it has to be DUPLEX because you
are going to write to it first, then read it and pipe it again to another writable
stream.
```