



CommonJS Modules

Learning Objectives

- Understand why dividing software into modules is important
- Understand how to use CommonJS modules

Material

Why Modules?

The best way to create complicated things is to assemble them from simpler units or “modules”. These units should be independent of one another as far as possible, as this makes it easier to modify individual modules without unexpected side-effects.

In the majority of programming languages, code in separate files does not interact with each other unless you explicitly export and import it. This is true of JavaScript running within Node.js but **not** within the browser. Take the following example:

```
<!DOCTYPE html>
  <body>
    <script src="line1.js"></script>
    <script src="line2.js"></script>
  </body>
</html>
```

```
// line1.js
const a = 5;
```

```
// line2.js
console.log(a);
```

If we open this html document as a web-page, we see that “5” is printed to the console: the code works. It’s as though line1.js and line2.js were part of the same file.

Almost all JavaScript developers agree that this behaviour is **bad**. We want to be able to write JavaScript in files without worrying about unanticipated interactions from code in other files.



CommonJS Modules

In Node.js, you have to explicitly export variables from a file in order for them to be imported and used by another. To illustrate this, let's imagine we're creating a weather forecast app:

```
weatherApp/  
├─ node_modules/  
│   └─ express/  
├─ utils/  
│   └─ converters.js  
├─ app.js  
├─ getForecast.js  
├─ package-lock.json  
└─ package.json
```

Our `getForecast.js` file has a function that `app.js` needs to use. Node.js, by default, uses the CommonJS module style so we can export a function using `module.exports` like so:

```
// getForecast.js  
function getForecast() {  
    /*  
     gets the weather forecast  
    */  
}  
module.exports = getForecast;
```

And import it using the `require` keyword, passing in the path to the `getForecast.js` file relative to the current file, like this:

```
// app.js  
const getForecast = require("./getForecast");  
  
const forecast = getForecast();  
  
// do stuff with forecast
```

The `./` indicates that the `getForecast.js` file is in the same folder as `app.js`.

We want our app to support both Fahrenheit and Celsius temperatures so we create two converter functions to use throughout our system. To export multiple entities we can use JavaScript's destructuring assignment:

```
// converters.js
```



```
function fahrenheitToCelsius(f) {  
    return ((f - 32) * 5) / 9;  
}  
  
function celsiusToFahrenheit(c) {  
    return (c * 9) / 5 + 32;  
}  
  
module.exports = { fahrenheitToCelsius, CelsiusToFahrenheit };
```

We can then import the two functions like this:

```
// app.js  
const {  
    fahrenheitToCelsius,  
    celsiusToFahrenheit,  
} = require("./utils/converters");
```

Notice how we use the same destructuring assignment when importing. We also have to change the path because the `converters.js` file is in the `utils` subfolder.

If we want to import a 3rd-party node module - e.g. a built-in module like `path` or one installed using `npm` - we simply pass its name to `require` (rather than a file path):

```
const express = require("express");
```

Core Assignments

Collect Them All

```
git clone https://github.com/charliemerrell/common-js-collect-them-all.git
```


This repository is incomplete: the `collectThemAll.js` file uses a set of functions which it doesn't currently have access to. Clone the repository and then, by adding only `require` and `module.exports` statements, complete the code so that you see all 5 functions executing properly when you run:

```
node collectThemAll.js
```



You might notice the `package.json` has `chalk` listed as a dependency but the project has no `node_modules` - can you remember the `npm` command to install **all** a project's dependencies?

Additional Resources

 [JavaScript Modules Past & Present](#)