

# Exploration — Modules in JavaScript

## Introduction

---



The concept of **modules** (<https://www.freecodecamp.org/news/javascript-modules-explained-with-examples/>) is provided by many programming languages for providing functionality that can be reused by other programs. The idea is that developers writing a module (a script in a file) can selectively make certain classes or functions available for use outside the module (in other files), while hiding the implementation details of those classes or functions inside the module (of the original file). During our explorations, we have seen two different ways to use functionality defined in other files. When we introduced Node, we used the function `require()` to import packages into our code. Later when studying React, we used `import` to use components defined in other files. In this exploration, we will explain the difference between these two features and also study how to define new modules for our own code.

## ES Modules Using Export and Import Keywords

---

JavaScript started out as a language to develop small programs that comprised a single file. For the longest of times, the language did not have any standard support for defining modules. It wasn't until 2015 that support for modules was added to the language standard with `export` and `import` keywords. This support was added in the 6th Edition of **the ECMAScript standard** (<https://en.wikipedia.org/wiki/ECMAScript>) (ES6 for short) which is an international standard for JavaScript. These modules are referred to as ECMA Modules or ES6 Modules or simply **ES Modules**. ES Modules are defined in files which have either the extension `.js` or the extension `.mjs`. The standard recommends using the extension `.mjs` and that's what we will use in our examples.

## Using export to Export Named Features

When we write code in an ES module, only those functions, classes or variables that are explicitly exported using the keyword `export` are available outside the module. Such exports are called **named exports**. Everything else is inaccessible to code outside the module. Note that only features defined at the top level of the module can be exported. Local functions, classes or variables cannot be exported. There are two different syntaxes to specify exports from a module.

### Using the export keyword in the declaration

We can specify exports by using the `export` keyword before the declaration of the function, class or variable.

#### Example: model.mjs

For example, consider a module in the file `model.mjs` contained the following functions, classes and variables. Only the function `readEntity()` and the variable `COUNTRY` will be available outside the module, while the function `createEntity`, the class `Entity` and the variable `STATE` will be inaccessible to code using this module.

```
function createEntity(x) {  
  return new Entity(x);  
}  
  
export function readEntity() {  
  return new Entity("This be the entity");  
}
```

```
class Entity {  
  constructor(name) {  
    this.name = name;  
  }  
}  
  
const STATE = 'TX';  
  
export const COUNTRY = 'USA';
```

## Using an export statement

We can specify the names of the exported features using an `export` statement. The following code exports exactly the same features as are exported in the previous example by using the `export` keyword in the declarations.

```
function createEntity(...){ ...}  
function readEntity(...){ ... }  
class Entity { ... }  
const STATE = 'TX';  
const COUNTRY = 'USA';  
  
export { readEntity, COUNTRY }
```

Using the `export` statement provides one additional functionality that we can provide different names for the exported features. For example, the following `export` statement exports the function `readEntity` with the name `selectEntity` and code using this module can access this function only using the name `selectEntity`.

```
export { readEntity as selectEntity, COUNTRY }
```

## Using import to Import Named Features

We import named features from a module using the `import` statement along with the name of the features we want to import and the URL of the file containing the module.

### Example: Importing from model.mjs

Consider we are writing a program in a file that is in the same directory where the file `model.mjs` from the earlier example is stored. We can import the two features from this file using the following `import` statement.

```
import { readEntity, COUNTRY } from './model.mjs'
```

## Details of Import Functionality

- We can select which features we want to import and there is no requirement that we import all the features from a module.
- We can import all the exported features into an object by using `*`.
  - For example, the statement `import * as Model from './model.mjs'` imports all features exported by `model.mjs`.
  - We can use these features in our program by prepending the feature with the name of the object, e.g., `Model.readEntity`.
- We can also rename the imported features in our program. We can use this functionality to prevent a name clash that would otherwise occur if we import features with the same name from two different modules.

## Example: Renaming Imports

The following statement renames the variable `COUNTRY` imported from `model.mjs` to `DEFAULT_COUNTRY`. The code using this import will access the variable using the name `DEFAULT_COUNTRY`.

```
import { readEntity, COUNTRY as DEFAULT_COUNTRY } from './model.mjs'
```

## Default Export and Import

We can tag at most one feature in an ES module with the `export default` tag. The benefit of default exports is that the syntax for importing the default feature is very simple.

### The Default Export

We can tag at most one function or class as `export default`. This can be done in addition to exporting any additional number of features using the named export syntax. For example, in the following example we have tagged the function `readEntity` as the default export.

```
function createEntity(...){ ...}  
export default function readEntity(...){ ... }  
class Entity { ... }  
const STATE = 'TX';  
export const COUNTRY = 'USA';
```

### Default Import

We import the default export of a module via the following syntax

```
import readEntity from './model.mjs'
```

Note that importing the default export does not require surrounding the feature with curly braces, i.e., `{}`. As opposed to this importing any named export requires that we surround the feature with curly braces.

We can choose the name with which we import the feature. E.g., if we wanted to use the name `selectEntity` for the default export from `model.mjs`, we would use the following import statement:

```
import selectEntity from './model.mjs'
```

We can also import both default and named exports from a module using a single statement, as in the following example:

```
import readEntity, {COUNTRY} from './model.mjs'
```

## ES Modules and strict mode.

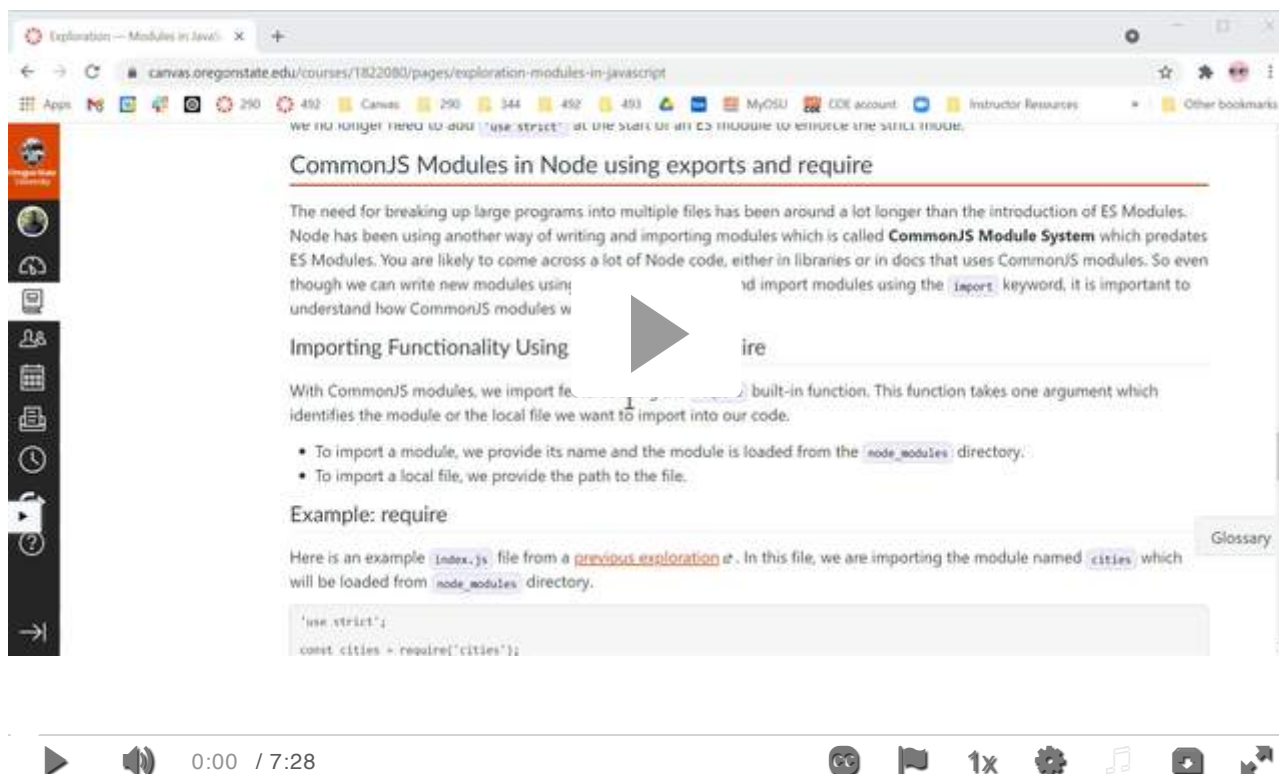
---

An additional feature that ES modules provide is that the entire contents of an ES module are **automatically in strict mode**. So we no longer need to add `'use strict'` at the start of an ES module to enforce the strict mode.

## CommonJS Modules in Node using exports and require()

---

The need for breaking up large programs into multiple files has been around a lot longer than the introduction of ES Modules. Node has been using another way of writing and importing modules which is called **CommonJS Module System** which predates ES Modules. You are likely to come across a lot of Node code, either in libraries or in docs that uses CommonJS modules. So even though we can write new modules using the `export` keyword and import modules using the `import` keyword, it is important to understand how CommonJS modules work.



The screenshot shows a video player interface. The video content is a webpage from canvas.oregonstate.edu titled "CommonJS Modules in Node using exports and require". The text on the page explains the CommonJS Module System and how to use the `require()` function to import modules. It includes an example code snippet:

```
'use strict';
const cities = require('cities');
let myCity = cities.zip_lookup('78704');
console.log(myCity);
```

The video player controls at the bottom show a play button, a volume icon, a progress bar at 0:00 / 7:28, and various settings icons.

## Importing Functionality Using the Function `require()`

With CommonJS modules, we import features using the `require()` built-in function. This function takes one argument which identifies the module or the local file we want to import into our code.

- To import a module, we provide its name and the module is loaded from the `node_modules` directory.
- To import a local file, we provide the path to the file.

### Example: `require`

Here is an example `index.js` file from a [previous exploration \(https://replit.com/@coecs290/m311#index.js\)](https://replit.com/@coecs290/m311#index.js). In this file, we are importing the module named `cities` which will be loaded from `node_modules` directory.

```
'use strict';
const cities = require('cities');
let myCity = cities.zip_lookup('78704');
console.log(myCity);
```

If we wanted to import a file named `model.js` which was in the same directory as the file which was importing it, we would use `require('./model.js')`.

## Exporting Functionality Using the `exports` Object

The function `require()` reads the module/file passed to it as an argument. It executes this module/file and returns the `exports` object from the file.

## Example: Using exports

Consider the following code was placed in a file `model.js`.

```
function createEntity(...){ ...}  
exports.readEntity = function(...){ ... }  
class Entity { ... }  
const STATE = 'TX';  
exports.country = 'USA';
```

Let us say, we import this file in our program using the following statement:

```
const myModel = require('./model.js');
```

The above statement will cause `model.js` to be read and evaluated. The `exports` object from `model.js` will be returned and it will be assigned to the variable `myModel`. The variable `myModel` will have two properties, `country` with the value "USA" and `readEntity` whose value will be the corresponding function defined in `model.js`.

```
{ country: "USA", readEntity: [Function]}
```

## Using ES Modules with Node

When running our code on Node, we can write ES modules and Node will support these modules. We can also import both ES and CommonJS modules using ES style `import` keyword. However to support ES module functionality, Node requires us to do either one of the following two things:

1. Use the extension `.mjs` for files which want to export modules using ES syntax and for files which want to import modules using ES syntax.
2. Use the extension `.js` for such files **and** also add a top level property "type" with the value "module" in the file `package.json`.

## Example: Using import for Express

We now rewrite our "Hello World" Express example from Module 3 using ES Module syntax in the following file `server.mjs`. Since the file is defined with the extension `.mjs`, Node will treat it as an ES Module. So instead of importing the express module using `require()`, we use `import` to import this module.

```
'use strict';
```



```
import express from 'express';

const app = express();
const PORT = 3000;

app.get("/", (req, res) => {
  res.send("Hello World!");
});

app.listen(PORT, () => {
  console.log(`Server listening on port ${PORT}...`);
});
```

What if we tried to import the express module using `require()`? For example, if we use the following statement in our `server.mjs` file, what will happen?

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

This will cause an error because Node treats a file with the `.mjs` extension as an ES module and the function `require()` is not available in an ES module.

## Summary

---

Modern JavaScript support defining modules using ES modules. In ES modules, we export classes, functions, and variables from a module using the `export` keyword. We can import the exported features into our ES modules using the `import` keyword. However, Node uses an older module system called CommonJS module system. In CommonJS modules export functionality using the `exports` object and functionality is imported using the `require` function.

When writing new code, we should use ES module functionality rather than CommonJS modules. But since a lot of existing code uses Node style CommonJS modules, it is important to understand how CommonJS modules work.

## Additional Resources

---

- **Chapter 10** [\\_\(https://eloquentjavascript.net/10\\_modules.html\)\\_](https://eloquentjavascript.net/10_modules.html) of Eloquent JavaScript has a good discussion of modules.
- JavaScript Modules are discussed at length **at MDN** [\\_\(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules\)\\_](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules).
- Node modules and the `require` function are described on the **Node website** [\\_\(https://nodejs.org/en/knowledge/getting-started/what-is-require/\)\\_](https://nodejs.org/en/knowledge/getting-started/what-is-require/) while the API doc for the `require` function is available **here** [\\_\(https://nodejs.org/docs/latest-v14.x/api/modules.html\)\\_](https://nodejs.org/docs/latest-v14.x/api/modules.html).