## What are Modules in Node.js?

In Node.js, a module is a collection of JavaScript functions and objects that can be used by external applications. Describing a piece of code as a module refers less to what the code is and more to what it does — any Node.js file or collection of files can be considered a module if its functions and data are made usable to external programs.

## Modules are useful for several reasons:

- They help you avoid naming conflicts by creating a separate scope for your variables and functions.
- They enable code reuse by allowing you to import and export functionality from other modules.
- They facilitate collaboration by letting you share your code with other developers through package managers like npm.

## Let’s take a look at how this is done with CommonJS.

### There are three types of modules in Node.js:

1. **Core modules -:**
   These are built-in modules that are part of the Node.js platform and come with the Node.js installation. They provide basic functionality such as file system access, network communication, cryptography, etc. You can load them by using the `require()` function with the module name as the argument. For example, `const fs = require('fs')` will load the file system module.

2. **Local modules**: These are custom modules that you create in your Node.js application. You can load them by using the `require()` function with the relative or absolute path to the module file as the argument. For example, `const math = require('./math.js')` will load the math module from the current directory.
3. **Third-party modules**: These are modules that are developed by other developers and published on npm, the Node.js package manager. You can install them by using the `npm install` command and load them by using the `require()` function with the module name as the argument. For example, `const express = require('express')` will load the express web framework module.

## How to Create a CommonJS in Node.js?

You can use the `module.exports` object or the `exports` shortcut to do this. For example, let's create a math module that exports two functions: `add()` and `multiply()`.

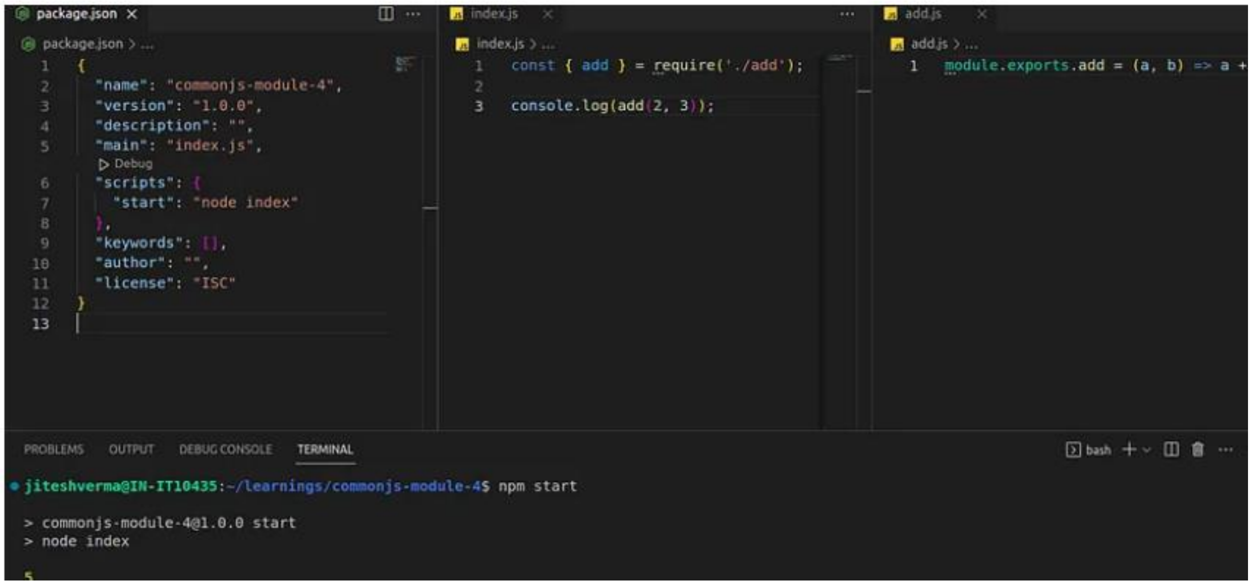
```
// math.js
// Define the add function
function add(a, b) {
  return a + b;
}
// Define the multiply function
function multiply(a, b) {
  return a * b;
}
// Export the functions
module.exports = {
  add: add,
  multiply: multiply
};
```

## How to Use a CommonJS Module in Node.js?

All you need to do is use the `require()` function to load the module and assign it to a variable.

```
// app.js
// Load the math module
const math = require('./math.js');
// Load the greeting module
const greeting = require('./greeting.js');
// Use the math module
console.log(math.add(2, 3)); // 5
console.log(math.multiply(2, 3)); // 6
// Use the greeting module
console.log(greeting('Ayush')); // Hello, Ayush
```

In the above code, we loaded the math and greeting modules by using the `require()` function with the relative paths to the module files as the arguments.



package.json

```json
{
  "name": "commonjs-module-4",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

index.js

```javascript
const { add } = require('./add');

console.log(add(2, 3));
```

add.js

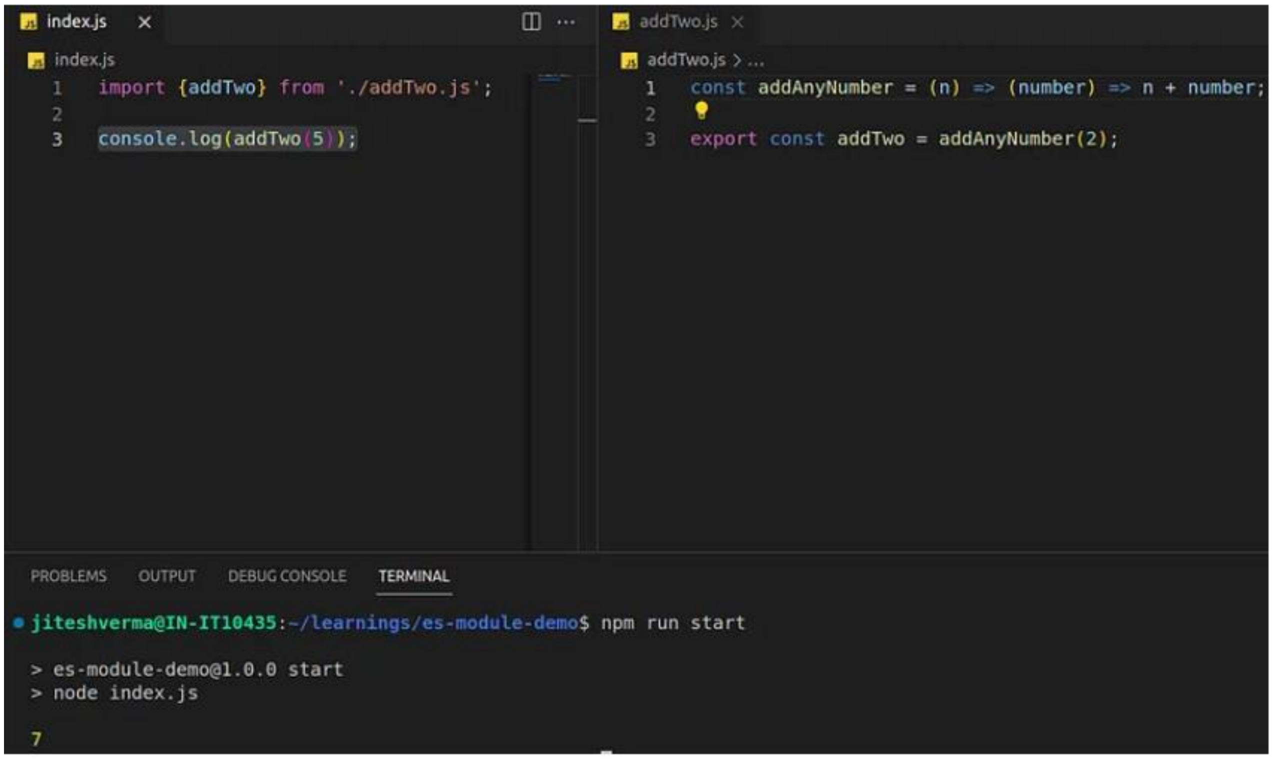
```javascript
module.exports.add = (a, b) => a +
```

```
jiteshverma@IN-IT10435:~/learnings/commonjs-module-4$ npm start

> commonjs-module-4@1.0.0 start
> node index

5
```

## Let’s take a look at how this is done with ES modules .

As of Node.js version 14.x, the standard JavaScript module system, ES modules, has also be adopted and provides alternatives to utilizing modules along with some key differences. The ES module system utilizes the “import” statement to import modules and does so asynchronously as opposed to the synchronous require() function.

standardized way of defining modules, allowing for more flexibility and cleaner code. They use the `import` and `export` statements to define and export module code and allow for named exports, default exports, and re-exports.



```javascript
import {addTwo} from './addTwo.js';

console.log(addTwo(5));
```

```javascript
const addAnyNumber = (n) => (number) => n + number;

export const addTwo = addAnyNumber(2);
```

```
jiteshverma@IN-IT10435:~/learnings/es-module-demo$ npm run start

> es-module-demo@1.0.0 start
> node index.js

7
```

## Pros and Cons of CommonJS and ES Modules

Following are the pros and cons of CommonJS modules:

- Module loading in CommonJS is synchronous. Because module loading is synchronous, it can block the main thread of execution and cause delays if the modules being loaded are large. It can pose rigorous performance issues for large-scale applications that can load hundreds of modules.
- In the case of legacy code or code written in the old version of NodeJS, CommonJS has full and stable support since it is the default standard. Versions preceding v8.5.0 have to be using this only.

Following are the pros and cons of ES modules:

- Module loading in ES Module is asynchronous by default. The asynchronous loading of ES modules can improve the performance of your code by allowing other code to continue executing while the module is being loaded.
- A file extension must be provided when using the `import` keyword. Directory indexes (e.g. `./directory/index.js`) must also be fully specified.
- We can import CommonJS modules from ES modules since, in this case, `module.exports` simply become the default export which we might import as such.