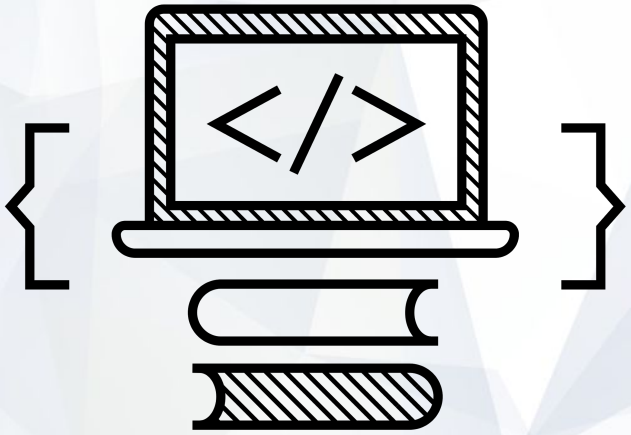
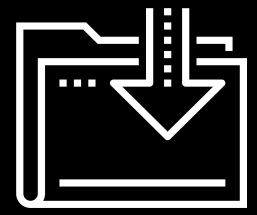


CommonJS

Course: Java

S1



Learning Outcomes

By the end of this lesson, you will be able to:

01 Describe the JavaScript approach to modules.

02 Break JavaScript code into multiple source files.

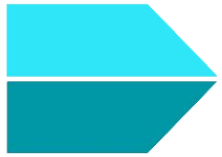
03 Create your own module.

04 Make an assignment to `module.exports` to export module code.

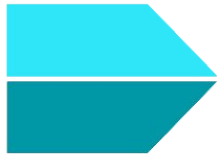
05 Import a module with `require`.

Modules

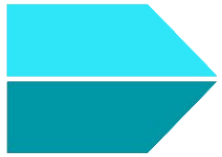
Module Concepts



A **module** is code that’s encapsulated in a file and exported to another file.



Modules focus on a single piece of functionality and stay loosely coupled with the other files in an application.



A module's definitions are private by default. We can’t access them unless they’re explicitly exported.

CommonJs

CommonJs is a module specification for server-side JavaScript.

It is the default module system for Node.js.

Other specifications include:

- **AMD:** An asynchronous module specification that's usually intended for the browser.
- **ES6:** The newest specification and a promising one, because it's part of the ECMAScript specification.

The module.exports Object

- In Node, `module.exports` is an in-scope JavaScript object. It works like any other JavaScript object. We can assign new properties to it, modify the properties (although by default, it's empty), and replace it entirely.
- Note that `module` is also an object.

```
// Each module is its own file.
const FIVE = 5;

// private function
function add(a, b) {
    return a + b;
}

function addFive(a) {
    return add(a, FIVE);
}

// public API
module.exports.FIVE = FIVE;
module.exports.addFive = addFive;
```

The require Function

- The `require` function works with both local file paths and module names. The Node.js require docs provide full details.
- The `require` function returns `module.exports`—whatever its value.
- It accepts a local file path or third-party module name and tries to load code from that.

```
const example = require("./example.js");


console.log(typeof example.FIVE);    // number
console.log(typeof example.addFive); // function


console.log(example.addFive(10));    // 15
```

Destructuring require

- We can destructure our exports to simplify names.
- To avoid the extra object identifier and dot operator--for example, in `example.FIVE`--destructure the object, as this code shows:

```
const { FIVE, addFive } = require("./example.js");

console.log(typeof FIVE);     // number
console.log(typeof addFive);  // function

console.log(addFive(10));    // 15
```

Destructuring with an Alias

We're not stuck with the names that a module gives us. If a name conflicts with our module, or we don't like a name, we can create an alias via object destructuring.

```
const {
    FIVE: five,
    addFive: plusFive
} = require("./example.js");

console.log(typeof five);     // number
console.log(typeof addFive);  // undefined
console.log(typeof plusFive); // function

console.log(plusFive(10));    // 15
```

Replacing module.exports

- Note that `exports` starts as an object, but it can be replaced.
- We can assign any valid JavaScript value to it—from a single Boolean value to a gigantic object graph.

```
const FIVE = 5;

// private function
function add(a, b) {
    return a + b;
}

function addFive(a) {
    return add(a, FIVE);
}

// replace the object with a function
  module.exports = addFive;
```

require a Replaced exports

- Note that the `require` function returns `exports` regardless of its value.
- It’s even possible to assign an undefined `exports`, but that wouldn’t be very useful.

```
const plusFive = require("./example.js");

console.log(typeof plusFive); // function
console.log(plusFive(10));    // 15
```


Break

Modular Refactor for Tic-Tac-Toe

Suggested Time:

30 Minutes

Recap

Learning Outcomes

By the end of this lesson, you will be able to:

01 Describe the JavaScript approach to modules.

02 Break JavaScript code into multiple source files.

03 Create your own module.

04 Make an assignment to `module.exports` to export module code.

05 Import a module with `require`.

Questions?