

# Javascript V: Modules & Bundlers

---

Pau Fernández   Rafa Genés   Jose L. Muñoz

Universitat Politècnica de Catalunya (UPC)

CommonJS Modules

ES Modules

Bundlers

# CommonJS Modules

## Single file utilities module

To implement the `circle` module, write a file `circle.js` with:

```
// Define some utility functions
function area(r) {
  return Math.PI * r ** 2;
}

function circumference(r) {
  return 2 * Math.PI * r;
}

// Export them to the outside
module.exports = {
  area,
  circumference,
};
```

### Isolated Global Context

Global context invisible to the outside.

You export things explicitly.

### Sequential execution

Loading is executing the module.

You can do any sort of computed initialization.

# You export a Javascript value

Exporting many functions

```
module.exports = {  
  area,  
  circumference,  
};
```

Exporting a single function:

```
module.exports = function () { /* ... */ };
```

Exporting a value:

```
module.exports = 42;
```

Exporting a class:

```
module.exports = class Animal { /* ... */ };
```

To use a module, load it with `require`:

```
const circle = require('./circle');
```

`require` returns the value you put into `module.exports`.

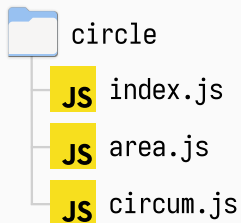
```
console.log(circle.area(1));  
console.log(circle.circumference(1));
```

Of course, destructuring works:

```
const { area, circumference } = require('./circle');
```

To write a module in a directory:

- Put the file modules in a directory.
- Write an `index.js` module as entry point.

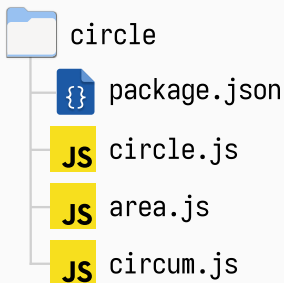




## Directory Modules with `package.json`

To write a module in a directory:

- Put the file modules in a directory.
- Create a `package.json` file.
- Specify the entry point with the `"main"` field.



### area.js

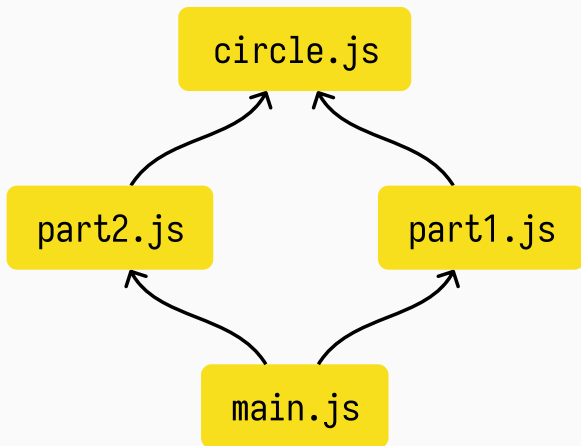
```
function area(r) {  
  return Math.PI * r**2;  
}  
module.exports = { area };
```

### circum.js

```
module.exports = function (r) {  
  return 2 * Math.PI * r;  
}
```

### circle.js

```
module.exports = {  
  area: require('./area').area,  
  circumference: require('./circum');  
}
```



The `package.json` resides at the base directory of a module and describes its properties:

- **name**: Name of the module.
- **version**: Version number.
- **description**: Textual description of the module.
- **main**: module ID that is the primary entry point.
- **dependencies**: Object that maps package names to version ranges.
- **private**: If the module is not to be published.

More properties: `homepage`, `directories`, `keywords`, `repository`, `bugs`, `license`, `files`, `browser`, `bin`, ...

Details: <https://docs.npmjs.com/files/package.json>

## Creating a `package.json`

To easily create a `package.json` for a new module:

```
npm init
```

This command will ask for:

- Package name
- Version
- Description
- Entry point (javascript file that will be loaded as the "main" file)
- Test command
- Git repository
- Keywords
- Author
- License

```
let mod = require('MODULE');
```

- 1) If `MODULE` is a core module, just load it.
- 2) If `MODULE` begins with `'./'` or `'../'`
  - a) Load as file (`'MODULE.js'`).
  - b) Load as directory:
    - b1) Parse `MODULE/package.json`, look for `"main"` field.
    - b2) Load `MODULE/<the file specified as "main">`.
- 3) Load from `node_modules` (either in the local directory or from any parent directory)

## Minimalistic **require** Implementation

This simplified implementation of **require** might throw some light about the process:

```
require.cache = Object.create(null);

function require(name) {
  if (!(name in require.cache)) {
    let code = readFile(name);
    let module = { exports: {} };
    require.cache[name] = module;
    let wrapper = Function("require, exports, module", code);
    wrapper(require, module.exports, module);
  }
  return require.cache[name].exports;
}
```

Before execution, modules are wrapped in a function that looks like this:

```
(function(require, exports, module, __filename, __dirname) {  
    // Module code actually lives in here  
});
```

This has the following consequences:

- Top-level variables are confined to the interior of the function and are thus local variables.
- It helps to provide some global-looking variables that are in fact specific to the module:
  - **module** and **exports** that the implementor can use to export values to the outside.
  - Convenience variables like **\_\_filename**, **\_\_dirname**.

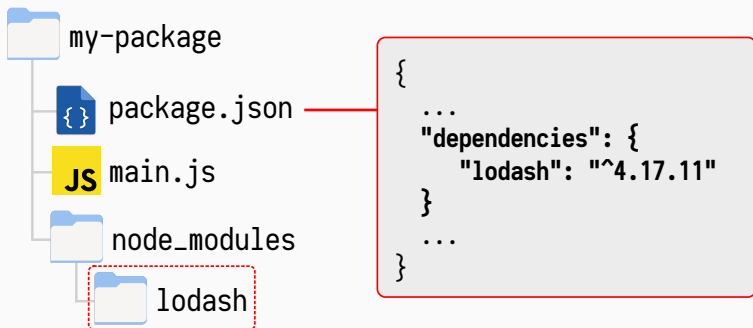


# Installing modules

Inside a Javascript package directory, installing a module is accomplished with:

```
npm install lodash
```

Two things happen: **a)** the module is installed into the `node_modules` local subdirectory and **b)** the dependency is registered in `package.json`:



A package-lock.json describes a particular `node_modules` tree (and associated `package.json` file), for the following purposes:

- Make things exactly reproducible: the `package-lock.json` will ensure that the `node_modules` folder installed by npm in different places is exactly the same.
- Provide a way to "time-travel": save the state of previous `node_modules` tree so that it is not necessary to save the whole tree.
- Make changes to the `node_modules` tree observable in diffs.
- Optimize npm module installation by caching metadata resolution for already installed packages.

Details: <https://docs.npmjs.com/files/package-lock.json>

NodeJS comes with core modules, implemented directly in the binary:

```
const nodejs_core_modules = {  
  os:    require('os'),      // Operating System  
  fs:    require('fs'),      // FileSystem (~POSIX)  
  http:  require('http'),    // HTTP servers/clients  
  https: require('https'),   // HTTP over TLS/SSL  
  net:   require('net'),     // TCP or IPC servers/clients  
  events: require('events'), // API for Emitters and Listeners  
  path:  require('path'),    // API for file and directory paths.  
  cprocs: require('child_processes'),  
                                     // Spawn child child_processes  
  // ...  
}
```

CommonJS Modules

ES Modules

Bundlers

# ES Modules

ESM is a standard which defines module loading as three steps:

1. Construction
2. Instantiation
3. Evaluation

This separation enables two important features:

- Asynchronous module loading.
- Tree-shaking.

The ESM standard also simplifies import/export syntax.

We can have several **exported** things per module:

```
// circle.js
export function area(r) {
  return Math.PI * r * r;
}
export function circumference(r) {
  return 2 * Math.PI * r;
}
export const tau = 2 * Math.PI;
```

---

```
// index.js
import { area, tau } from './circle';
```

The names are important, we will use them when importing.

Write code as usual, then mark things you want to **export**.

Different **exports** can be **imported** selectively.

Selective importing allows bundlers to do "tree-shaking".



## Single **export** statement

Instead of marking everything with **export**, we can issue a single **export** statement at the end:

```
// circle.js  
function area(r) {  
  return Math.PI * r * r;  
}  
function circumference(r) {  
  return 2 * Math.PI * r;  
}  
const tau = 2 * Math.PI;  
  
// We choose here what to export  
export { area, circumference, tau };
```

## Import all named exports

To **import** all functions at once, we use **\*** and name the object that will contain all the imported functions:

```
// circle.js  
export const area = (r) => Math.PI * r * r;  
export const circumference = (r) => 2 * Math.PI * r;
```

---

```
// index.js  
import * as circFuncs from './circle';  
console.log(circFuncs.area(4.5));
```

## default export

One special thing can be marked as the **default** export

```
// my-component.js
export default class MyComponent {
  // ...
}
```

The default export does not need a name (it is an expression, actually).

The default export is **independent** of named exports.

---

Importing the default thing in a module has the simplest syntax:

```
// index.js
import MyComponent from './my-component';
```

## Renaming when importing

```
// circle.js  
export const circleArea = (r) => Math.PI * r * r;  
export const circleCircum = (r) => 2 * Math.PI * r;
```

---

Named imports can be renamed when importing

```
// index.js  
import {  
  circleArea as circ,  
  circleCircum as circumference  
} from './circle';
```

## Renaming when exporting

Or renamed when exporting

```
// circle.js
const circleArea = (r) => Math.PI * r * r;
const circleCirc = (r) => 2 * Math.PI * r;

export {
  circleArea as area,
  circleCirc as circumference
};
```

---

```
// index.js
import { area, circumference } from './circle';
```

To write modules in different files, you can directly export imported things:

```
// Export all named symbols from "alpha"  
export * from 'alpha';
```

```
// Export specific things from "beta"  
export { a, b } from 'beta';
```

```
// Export specific renamed things from "beta"  
export { a as aaa, b as bbb } from 'beta';
```

```
// Export the default from "gamma"  
export { default } from 'gamma';
```

```
// Export a named thing as the default from "delta"  
export { c as default } from 'delta';
```

```
// util.js
const removeAllChildren = (elem) => {
  while (elem.firstChild) {
    elem.firstChild.remove();
  }
}
```

The `type="module"` attribute tells the browser a script is an ES module.

```
<!-- index.html -->
<script type="module">
  import { removeAllChildren } from './util.js';
  removeAllChildren(document.body);
</script>
<!-- ... -->
```

Importing from other modules loads them relative to the current URL.

Two ways to use ES Modules in Node (version > 12):

- Files named with an `.mjs` extension are considered ES modules.
- Directory modules with a `"type": "module"` entry in `package.json` are also considered ES modules.



CommonJS Modules

ES Modules

Bundlers

Bundlers

## Input files

HTML, CSS, Javascript, images (pngs, jpg, svg, ...), icons, fonts, etc.

## Output files

Processed input files:

- Transpile Javascript (ES6+ to ES5).
- Compile SCSS to CSS.
- Put images in specific folders (+ generate URL).
- Minify Javascript (compress).
- Tree shake (remove unused code).
- Split into chunks (faster loading).

Bundlers automate front-end compilation:

- They read dependencies from **imports** (including images and JSON).
- Create a dependency graph (who needs whom).
- Put output files in specific folders.
- Compile all code into a single Javascript file (the **bundle**).

The most popular bundler nowadays is **webpack**.

Webpack has some interesting features:

- Loaders: read many different file formats (plugins).
- Transformers: read and process files (plugins).
- Code-splitting (lazy-loading chunks).
- Hot-Module Replacement.

# Webpack demo

Create a directory and `package.json` inside:

```
npm init -y
```

Install `webpack` as a library and CLI:

```
npm install -D webpack webpack-cli
```

Create `index.js` and `dist/index.html`

Add scripts to `package.json`

```
{  
  "scripts": {  
    "build": "webpack --mode production"  
  }  
}
```

Execute `webpack`

```
npm run build
```

# Webpack configuration

If the file `webpack.config.js` exists, it is executed and exports a configuration object:

```
// webpack.config.js
const path = require('path');

module.exports = {
  // Entry point
  entry: './src/index.js',
  output: {
    filename: 'main.js', // Generated bundle name
    path: path.resolve(__dirname, 'dist') // Output directory
  }
};
```