# ECMAScript 6 modules: the final syntax

[2014-09-07] esnext, dev, javascript, jsmodules



**Check out my book (free online): "Exploring ES6".** Updated version of this blog post: chapter "Modules".

---

At the end of July 2014, TC39 [1] had another meeting, during which the last details of the ECMAScript 6 (ES6) module syntax were finalized. This blog post gives an overview of the complete ES6 module system.

## 1. Module systems for current JavaScript

JavaScript does not have built-in support for modules, but the community has created impressive work-arounds. The two most important (and unfortunately incompatible) standards are:

- **CommonJS Modules:** The dominant implementation of this standard is in Node.js (Node.js modules have a few features that go beyond CommonJS). Characteristics:
  - Compact syntax
  - Designed for synchronous loading
  - Main use: server
- **Asynchronous Module Definition (AMD):** The most popular implementation of this standard is RequireJS. Characteristics:
  - Slightly more complicated syntax, enabling AMD to work without eval() (or a compilation step).
  - Designed for asynchronous loading
  - Main use: browsers

The above is but a grossly simplified explanation of the current state of affairs. If you want more in-depth material, take a look at "Writing Modular JavaScript With AMD, CommonJS & ES Harmony" by Addy Osmani.

## 2. ECMAScript 6 modules

The goal for ECMAScript 6 modules was to create a format that both users of CommonJS and of AMD are happy with:

- Similar to CommonJS, they have a compact syntax, a preference for single exports and support for cyclic dependencies.
- Similar to AMD, they have direct support for asynchronous loading and configurable module loading.

Being built into the language allows ES6 modules to go beyond CommonJS and AMD (details are explained later):

- Their syntax is even more compact than CommonJS's.
- Their structure can be statically analyzed (for static checking, optimization, etc.).
- Their support for cyclic dependencies is better than CommonJS's.

The ES6 module standard has two parts:

- Declarative syntax (for importing and exporting)
- Programmatic loader API: to configure how modules are loaded and to conditionally load modules

## 3. An overview of the ES6 module syntax

There are two kinds of exports: named exports (several per module) and default exports (one per module).

### 3.1. Named exports (several per module)

A module can export multiple things by prefixing their declarations with the keyword `export`. These exports are distinguished by their names and are called *named exports*.

```
//------ lib.js ------
export const sqrt = Math.sqrt;
export function square(x) {
    return x * x;
}
export function diag(x, y) {
    return sqrt(square(x) + square(y));
}

//------ main.js ------
import { square, diag } from 'lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

There are other ways to specify named exports (which are explained later), but I find this one quite convenient: simply write your code as if there were no outside world, then label everything that you want to export with a keyword.

If you want to, you can also import the whole module and refer to its named exports via property notation:

```
//------ main.js ------
import * as lib from 'lib';
console.log(lib.square(11)); // 121
console.log(lib.diag(4, 3)); // 5
```

**The same code in CommonJS syntax:** For a while, I tried several clever strategies to be less redundant with my module exports in Node.js. Now I prefer the following simple but slightly verbose style that is reminiscent of the revealing module pattern:

```
//------ lib.js ------
var sqrt = Math.sqrt;
function square(x) {
    return x * x;
}
function diag(x, y) {
    return sqrt(square(x) + square(y));
}
module.exports = {
    sqrt: sqrt,
    square: square,
    diag: diag,
};

//------ main.js ------
var square = require('lib').square;
var diag = require('lib').diag;
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

### 3.2. Default exports (one per module)

Modules that only export single values are very popular in the Node.js community. But they are also common in frontend development where you often have constructors/classes for models, with one model per module. An ECMAScript 6 module can pick a *default export*, the most important exported value. Default exports are especially easy to import.

The following ECMAScript 6 module "is" a single function:

```
//------ myFunc.js ------
export default function () { ... };

//------ main1.js ------
import myFunc from 'myFunc';
myFunc();
```

An ECMAScript 6 module whose default export is a class looks as follows:

```
//------ MyClass.js ------
export default class { ... };

//------ main2.js ------
import MyClass from 'MyClass';
let inst = new MyClass();
```

Note: The operand of the default export declaration is an expression, it often does not have a name. Instead, it is to be identified via its module's name.

### 3.3. Having both named exports and a default export in a module

The following pattern is surprisingly common in JavaScript: A library is a single function, but additional services are provided via properties of that function. Examples include jQuery and Underscore.js. The following is a sketch of Underscore as a CommonJS module:

```
//------ underscore.js ------
var _ = function (obj) {
    ...
};
var each = _.each = _.forEach =
    function (obj, iterator, context) {
        ...
    };
module.exports = _;

//------ main.js ------
var _ = require('underscore');
var each = _.each;
...
```

With ES6 glasses, the function _ is the default export, while `each` and `forEach` are named exports. As it turns out, you can actually have named exports and a default export at the same time. As an example, the previous CommonJS module, rewritten as an ES6 module, looks like this:

```
//------ underscore.js ------
export default function (obj) {
    ...
};
export function each(obj, iterator, context) {
    ...
}
export { each as forEach };

//------ main.js ------
import _, { each } from 'underscore';
...
```

Note that the CommonJS version and the ECMAScript 6 version are only roughly similar. The latter has a flat structure, whereas the former is nested. Which style you prefer is a matter of taste, but the flat style has the advantage of being statically analyzable (why that is good is explained below). The CommonJS style seems partially motivated by the need for objects as namespaces, a need that can often be fulfilled via ES6 modules and named exports.

#### The default export is just another named export

The default export is actually just a named export with the special name `default`. That is, the following two statements are equivalent:

```
import { default as foo } from 'lib';
import foo from 'lib';
```

Similarly, the following two modules have the same default export:

```
//------ module1.js ------
export default 123;

//------ module2.js ------
const D = 123;
export { D as default };
```

#### Why do we need named exports?

You may be wondering – why do we need named exports if we could simply default-export objects (like CommonJS)? The answer is that you can't enforce a static structure via objects and lose all of the associated advantages (described in the next section).

## 4. Design goals

If you want to make sense of ECMAScript 6 modules, it helps to understand what goals influenced their design. The major ones are:

- Default exports are favored
- Static module structure
- Support for both synchronous and asynchronous loading
- Support for cyclic dependencies between modules

The following subsections explain these goals.

### 4.1. Default exports are favored

The module syntax suggesting that the default export "is" the module may seem a bit strange, but it makes sense if you consider that one major design goal was to make default exports as convenient as possible. Quoting David Herman:

> ECMAScript 6 favors the single/default export style, and gives the sweetest syntax to importing the default. Importing named exports can and even should be slightly less concise.

### 4.2. Static module structure

In current JavaScript module systems, you have to execute the code in order to find out what the imports and exports are. That is the main reason why ECMAScript 6 breaks with those systems: by building the module system into the language, you can syntactically enforce a static module structure. Let's first examine what that means and then what benefits it brings.

A module's structure being static means that you can determine imports and exports at compile time (statically) – you only have to look at the source code, you don't have to execute it. The following are two examples of how CommonJS modules can make that impossible. In the first example, you have to run the code to find out what it imports:

```
var mylib;
if (Math.random()) {
    mylib = require('foo');
} else {
    mylib = require('bar');
}
```

In the second example, you have to run the code to find out what it exports:

```
if (Math.random()) {
    exports.baz = ...;
}
```

ECMAScript 6 gives you less flexibility, it forces you to be static. As a result, you get several benefits [2], which are described next.

#### Benefit 1: faster lookup

If you require a library in CommonJS, you get back an object:

```
var lib = require('lib');
lib.someFunc(); // property lookup
```

Thus, accessing a named export via `lib.someFunc` means you have to do a property lookup, which is slow, because it is dynamic.

In contrast, if you import a library in ES6, you statically know its contents and can optimize accesses:

```
import * as lib from 'lib';
lib.someFunc(); // statically resolved
```

#### Benefit 2: variable checking

With a static module structure, you always statically know which variables are visible at any location inside the module:

- Global variables: increasingly, the only completely global variables will come from the language proper. Everything else will come from modules (including functionality from the standard library and the browser). That is, you statically know all global variables.
- Module imports: You statically know those, too.
- Module-local variables: can be determined by statically examining the module.

This helps tremendously with checking whether a given identifier has been spelled properly. This kind of check is a popular feature of linters such as JSLint and JSHint; in ECMAScript 6, most of it can be performed by JavaScript engines.

Additionally, any access of named imports (such as `lib.foo`) can also be checked statically.

#### Benefit 3: ready for macros

Macros are still on the roadmap for JavaScript's future. If a JavaScript engine supports macros, you can add new syntax to it via a library. Sweet.js is an experimental macro system for JavaScript. The following is an example from the Sweet.js website: a macro for classes.

```
// Define the macro
macro class {
    rule {
        $className {
                constructor $cparams $cbody
                $($mname $mparams $mbody) ...
        }
    } => {
        function $className $cparams $cbody
        $($className.prototype.$mname
            = function $mname $mparams $mbody; ) ...
    }
}

// Use the macro
class Person {
    constructor(name) {
        this.name = name;
    }
    say(msg) {
        console.log(this.name + " says: " + msg);
    }
}
var bob = new Person("Bob");
bob.say("Macros are sweet!");
```

For macros, a JavaScript engine performs a preprocessing step before compilation: If a sequence of tokens in the token stream produced by the parser matches the pattern part of the macro, it is replaced by tokens generated via the body of macro. The preprocessing step only

works if you are able to statically find macro definitions. Therefore, if you want to import macros via modules then they must have a static structure.

#### Benefit 4: ready for types

Static type checking imposes constraints similar to macros: it can only be done if type definitions can be found statically. Again, types can only be imported from modules if they have a static structure.

Types are appealing because they enable statically typed fast dialects of JavaScript in which performance-critical code can be written. One such dialect is Low-Level JavaScript (LLJS). It currently compiles to asm.js.

#### Benefit 5: supporting other languages

If you want to support compiling languages with macros and static types to JavaScript then JavaScript's modules should have a static structure, for the reasons mentioned in the previous two sections.

### 4.3. Support for both synchronous and asynchronous loading

ECMAScript 6 modules must work independently of whether the engine loads modules synchronously (e.g. on servers) or asynchronously (e.g. in browsers). Its syntax is well suited for synchronous loading, asynchronous loading is enabled by its static structure: Because you can statically determine all imports, you can load them before evaluating the body of the module (in a manner reminiscent of AMD modules).

### 4.4. Support for cyclic dependencies between modules

Two modules A and B are cyclically dependent on each other if both A (possibly indirectly/transitively) imports B and B imports A. If possible, cyclic dependencies should be avoided, they lead to A and B being *tightly coupled* – they can only be used and evolved together.

#### Why support cyclic dependencies?

Cyclic dependencies are not inherently evil. Especially for objects, you sometimes even want this kind of dependency. For example, in some trees (such as DOM documents), parents refer to children and children refer back to parents. In libraries, you can usually avoid cyclic dependencies via careful design. In a large system, though, they can happen, especially during refactoring. Then it is very useful if a module system supports them, because then the system doesn't break while you are refactoring.

The Node.js documentation acknowledges the importance of cyclic dependencies [3] and Rob Sayre provides additional evidence:

> Data point: I once implemented a system like [ECMAScript 6 modules] for Firefox. I got asked for cyclic dependency support 3 weeks after shipping.
>
> That system that Alex Fritze invented and I worked on is not perfect, and the syntax isn't very pretty. But it's still getting used 7 years later, so it must have gotten something right.

Let's see how CommonJS and ECMAScript 6 handle cyclic dependencies.

#### Cyclic dependencies in CommonJS

In CommonJS, if a module B requires a module A whose body is currently being evaluated, it gets back A's exports object in its current state (line #1 in the following example). That enables B to refer to properties of that object inside its exports (line #2). The properties are filled in after B's evaluation is finished, at which point B's exports work properly.

```
//------ a.js ------
var b = require('b');
exports.foo = function () { ... };

//------ b.js ------
var a = require('a'); // (1)
// Can't use a.foo in module body,
// but it will be filled in later
exports.bar = function () {
    a.foo(); // OK (2)
};

//------ main.js ------
var a = require('a');
```

As a general rule, keep in mind that with cyclic dependencies, you can't access imports in the body of the module. That is inherent to the

phenomenon and doesn't change with ECMAScript 6 modules.

The limitations of the CommonJS approach are:

- Node.js-style single-value exports don't work. In Node.js, you can export single values instead of objects, like this:
  `module.exports = function () { ... }`
  If you did that in module A, you wouldn't be able to use the exported function in module B, because B's variable a would still refer to A's original exports object.
- You can't use named exports directly. That is, module B can't import `a.foo` like this:
  `var foo = require('a').foo;`
  `foo` would simply be `undefined`. In other words, you have no choice but to refer to `foo` via the exports object `a`.

CommonJS has one unique feature: you can export before importing. Such exports are guaranteed to be accessible in the bodies of importing modules. That is, if A did that, they could be accessed in B's body. However, exporting before importing is rarely useful.

#### Cyclic dependencies in ECMAScript 6

In order to eliminate the aforementioned two limitations, ECMAScript 6 modules export bindings, not values. That is, the connection to variables declared inside the module body remains live. This is demonstrated by the following code.

```
//------ lib.js ------
export let counter = 0;
export function inc() {
    counter++;
}

//------ main.js ------
import { inc, counter } from 'lib';
console.log(counter); // 0
inc();
console.log(counter); // 1
```

Thus, in the face of cyclic dependencies, it doesn't matter whether you access a named export directly or via its module: There is an indirection involved in either case and it always works.

## 5. More on importing and exporting

### 5.1. Importing

ECMAScript 6 provides the following ways of importing [4]:

```
// Default exports and named exports
import theDefault, { named1, named2 } from 'src/mylib';
import theDefault from 'src/mylib';
import { named1, named2 } from 'src/mylib';

// Renaming: import named1 as myNamed1
import { named1 as myNamed1, named2 } from 'src/mylib';

// Importing the module as an object
// (with one property per named export)
import * as mylib from 'src/mylib';

// Only load the module, don't import anything
import 'src/mylib';
```

### 5.2. Exporting

There are two ways in which you can export things that are inside the current module [5]. On one hand, you can mark declarations with the keyword `export`.

```
export var myVar1 = ...;
export let myVar2 = ...;
export const MY_CONST = ...;

export function myFunc() {
    ...
}
export function* myGeneratorFunc() {
    ...
}
export class MyClass {
    ...
}
```

The “operand” of a default export is an expression (including function expressions and class expressions). Examples:

```
export default 123;
export default function (x) {
    return x
};
export default x => x;
export default class {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
};
```

On the other hand, you can list everything you want to export at the end of the module (which is once again similar in style to the revealing module pattern).

```
const MY_CONST = ...;
function myFunc() {
    ...
}

export { MY_CONST, myFunc };
```

You can also export things under different names:

```
export { MY_CONST as THE_CONST, myFunc as theFunc };
```

Note that you can’t use reserved words (such as `default` and `new`) as variable names, but you can use them as names for exports (you can also use them as property names in ECMAScript 5). If you want to directly import such named exports, you have to rename them to proper variables names.

### 5.3. Re-exporting

Re-exporting means adding another module’s exports to those of the current module. You can either add all of the other module’s exports:

```
export * from 'src/other_module';
```

Or you can be more selective (optionally while renaming):

```
export { foo, bar } from 'src/other_module';

// Export other_module’s foo as myFoo
export { foo as myFoo, bar } from 'src/other_module';
```

## 6. eval() and modules

`eval()` does not support module syntax. It parses its argument according to the Script grammar rule and scripts don't support module syntax (why is explained later). If you want to evaluate module code, you can use the module loader API (described next).

## 7. The ECMAScript 6 module loader API

In addition to the declarative syntax for working with modules, there is also a programmatic API. It allows you to:

- Programmatically work with modules and scripts
- Configure module loading

Loaders handle resolving *module specifiers* (the string IDs at the end of `import...from`), loading modules, etc. Their constructor is `Reflect.Loader`. Each platform keeps a customized instance in the global variable `System` (the *system loader*), which implements its specific style of module loading.

### 7.1. Importing modules and loading scripts

You can programmatically import a module, via an API based on ES6 promises:

```
System.import('some_module')
.then(some_module => {
    // Use some_module
})
.catch(error => {
    ...
});
```

`System.import()` enables you to:

- Use modules inside `<script>` elements (where module syntax is not supported, consult Sect. "Further information" for details).
- Load modules conditionally.

`System.import()` retrieves a single module, you can use `Promise.all()` to import several modules:

```
Promise.all(
    ['module1', 'module2', 'module3']
    .map(x => System.import(x)))
.then(([module1, module2, module3]) => {
    // Use module1, module2, module3
});
```

More loader methods:

- `System.module(source, options?)` evaluates the JavaScript code in `source` to a module (which is delivered asynchronously via a promise).
- `System.set(name, module)` is for registering a module (e.g. one you have created via `System.module()`).
- `System.define(name, source, options?)` both evaluates the module code in `source` and registers the result.

### 7.2. Configuring module loading

The module loader API has various hooks for configuration. It is still work in progress. A first system loader for browsers is currently being implemented and tested. The goal is to figure out how to best make module loading configurable.

The loader API will permit many customizations of the loading process. For example:

1. Lint modules on import (e.g. via JSLint or JSHint).
2. Automatically translate modules on import (they could contain CoffeeScript or TypeScript code).
3. Use legacy modules (AMD, Node.js).

Configurable module loading is an area where Node.js and CommonJS are limited.

## 8. Further information

The following content answers two important questions related to ECMAScript 6 modules: How do I use them today? How do I embed them in HTML?

- **"Using ECMAScript 6 today"** gives an overview of ECMAScript 6 and explains how to compile it to ECMAScript 5. If you are interested in

the latter, start reading in Sect. 2. One intriguing minimal solution is the ES6 Module Transpiler which only adds ES6 module syntax to ES5 and compiles it to either AMD or CommonJS.

- **Embedding ES6 modules in HTML:** The code inside `<script>` elements does not support module syntax, because the element's synchronous nature is incompatible with the asynchronicity of modules. Instead, you need to use the new `<module>` element. The blog post "ECMAScript 6 modules in future browsers" explains how `<module>` works. It has several significant advantages over `<script>` and can be polyfilled in its alternative version `<script type="module">`.
- **CommonJS vs. ES6:** "JavaScript Modules" (by Yehuda Katz) is a quick intro to ECMAScript 6 modules. Especially interesting is a second page where CommonJS modules are shown side by side with their ECMAScript 6 versions.

## 9. Benefits of ECMAScript 6 modules

At first glance, having modules built into ECMAScript 6 may seem like a boring feature – after all, we already have several good module systems. But ECMAScript 6 modules have features that you can't add via a library, such as a very compact syntax and a static module structure (which helps with optimizations, static checking and more). They will also – hopefully – end the fragmentation between the currently dominant standards CommonJS and AMD.

Having a single, native standard for modules means:

- No more UMD (Universal Module Definition): UMD is a name for patterns that enable the same file to be used by several module systems (e.g. both CommonJS and AMD). Once ES6 is the only module standard, UMD becomes obsolete.
- New browser APIs become modules instead of global variables or properties of `navigator`.
- No more objects-as-namespaces: Objects such as `Math` and `JSON` serve as namespaces for functions in ECMAScript 5. In the future, such functionality can be provided via modules.

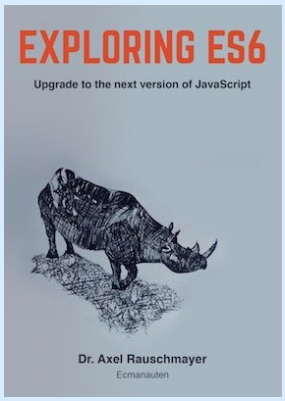
**Acknowledgements:** Thanks to Domenic Denicola for confirming the final module syntax. Thanks for corrections of this blog post go to: Guy Bedford, John K. Paul, Mathias Bynens, Michael Ficarra.

## 10. References

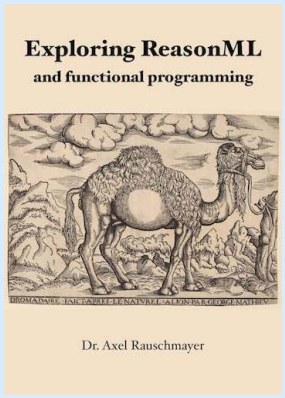
1. A JavaScript glossary: ECMAScript, TC39, etc. ↩
2. "Static module resolution" by David Herman ↩
3. "Modules: Cycles" in the Node.js API documentation ↩
4. "Imports" (ECMAScript 6 specification) ↩
5. "Exports" (ECMAScript 6 specification) ↩

Dr. Axel Rauschmayer
Twitter


Free ES.next newsletter




Book (free online)




Book (free online)

## Most popular (last 30 days)

Loading...