16. Modules

16.1 Overview

JavaScript has had modules for a long time. However, they were implemented via libraries, not built into the language. ES6 is the first time that JavaScript has built-in modules.

ES6 modules are stored in files. There is exactly one module per file and one file per module. You have two ways of exporting things from a module. These two ways can be mixed, but it is usually better to use them separately.

16.1.1 Multiple named exports

There can be multiple *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
```

You can also import the complete module:

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

16.1.2 Single default export

There can be a single *default export*. For example, a function:

```
//------ myFunc.js ------
export default function () { ··· } // no semicolon!

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

Or a class:

```
//------ MyClass.js ------
export default class { ··· } // no semicolon!

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

Note that there is no semicolon at the end if you default-export a function or a class (which are anonymous declarations).

16.1.3 Browsers: scripts versus modules

	Scripts	**Modules**
HTML element	`<script>`	`<script type="module">`
Default mode	non-strict	strict
Top-level variables are	global	local to module
Value of `this` at top level	`window`	`undefined`
Executed	synchronously	asynchronously
Declarative imports (`import` statement)	no	yes
Programmatic imports (Promise-based API)	yes	yes
File extension	`.js`	`.js`

16.2 Modules in JavaScript

Even though JavaScript never had built-in modules, the community has converged on a simple style of modules, which is supported by libraries in ES5 and earlier. This style has also been adopted by ES6:

- Each module is a piece of code that is executed once it is loaded.
- In that code, there may be declarations (variable declarations, function declarations, etc.).
    - By default, these declarations stay local to the module.
    - You can mark some of them as exports, then other modules can import them.
- A module can import things from other modules. It refers to those modules via *module specifiers*, strings that are either:
    - Relative paths (`'../model/user'`): these paths are interpreted relatively to the location of the importing module. The file extension `.js` can usually be omitted.

    - Absolute paths ('/lib/js/helpers'): point directly to the file of the module to be imported.
    - Names ('util'): What modules names refer to has to be configured.
- Modules are singletons. Even if a module is imported multiple times, only a single “instance” of it exists.

This approach to modules avoids global variables, the only things that are global are module specifiers.

16.2.1 ECMAScript 5 module systems

It is impressive how well ES5 module systems work without explicit support from the language. The two most important (and unfortunately incompatible) standards are:

- **CommonJS Modules:** The dominant implementation of this standard isin Node.js (Node.js modules have a few features that go beyond CommonJS). Characteristics:
    - Compact syntax
    - Designed for synchronous loading and servers
- **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 and browsers

The above is but a simplified explanation of ES5 modules. If you want more in-depth material, take a look at “Writing Modular JavaScript With AMD, CommonJS & ES Harmony” by Addy Osmani.

16.2.2 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:

- Similarly to CommonJS, they have a compact syntax, a preference for single exports and support for cyclic dependencies.
- Similarly 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

16.3 The basics of ES6 modules

There are two kinds of exports: named exports (several per module) and default exports (one per module). As explained later, it is possible use both at the same time, but usually best to keep them separate.

16.3.1 Named exports (several per module)

A module can export multiple things by prefixing its 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
```

16.3.2 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 classes for models and components, with one class per module. An ES6 module can pick a *default export*, the main exported value. Default exports are especially easy to import.

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

```
//------ myFunc.js ------
export default function () {} // no semicolon!

//------ 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 {} // no semicolon!

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

There are two styles of default exports:

1. Labeling declarations
2. Default-exporting values directly

16.3.2.1 Default export style 1: labeling declarations

You can prefix any function declaration (or generator function declaration) or class declaration with the keywords `export default` to make it the default export:

```
export default function foo() {} // no semicolon!
export default class Bar {} // no semicolon!
```

You can also omit the name in this case. That makes default exports the only place where JavaScript has anonymous function declarations and anonymous class declarations:

```
export default function () {} // no semicolon!
export default class {} // no semicolon!
```

16.3.2.1.1 Why anonymous function declarations and not anonymous function expressions?

When you look at the previous two lines of code, you'd expect the operands of `export default` to be expressions. They are only declarations for reasons of consistency: operands can be named declarations, interpreting their anonymous versions as expressions would be confusing (even more so than introducing new kinds of declarations).

If you want the operands to be interpreted as expressions, you need to use parentheses:

```
export default (function () {});
export default (class {});
```

16.3.2.2 Default export style 2: default-exporting values directly

The values are produced via expressions:

```
export default 'abc';
export default foo();
export default /^xyz$/;
export default 5 * 7;
export default { no: false, yes: true };
```

Each of these default exports has the following structure.

```
export default «expression»;
```

That is equivalent to:

```
const __default__ = «expression»;
export { __default__ as default }; // (A)
```

The statement in line A is an *export clause* (which is explained in a later section).

16.3.2.2.1 Why two default export styles?

The second default export style was introduced because variable declarations can't be meaningfully turned into default exports if they declare multiple variables:

```
export default const foo = 1, bar = 2, baz = 3; // not legal JavaScript!
```

Which one of the three variables `foo`, `bar` and `baz` would be the default export?

16.3.3 Imports and exports must be at the top level

As explained in more detail later, the structure of ES6 modules is *static*, you can't conditionally import or export things. That brings a variety of benefits.

This restriction is enforced syntactically by only allowing imports and exports at the top level of a module:

```
if (Math.random()) {
    import 'foo'; // SyntaxError
}

// You can't even nest `import` and `export`
// inside a simple block:
{
    import 'foo'; // SyntaxError
}
```

16.3.4 Imports are hoisted

Module imports are hoisted (internally moved to the beginning of the current scope).

Therefore, it doesn’t matter where you mention them in a module and the following code works without any problems:

```
foo();

import { foo } from 'my_module';
```

16.3.5 Imports are read-only views on exports

The imports of an ES6 module are read-only views on the exported entities. That means that the connections to variables declared inside module bodies remain live, as demonstrated in the following code.

```
//------ lib.js ------
export let counter = 3;
export function incCounter() {
    counter++;
}

//------ main.js ------
import { counter, incCounter } from './lib';

// The imported value `counter` is live
console.log(counter); // 3
incCounter();
console.log(counter); // 4
```

How that works under the hood is explained in a later section.

Imports as views have the following advantages:

- They enable cyclic dependencies, even for unqualified imports (as explained in the next section).
- Qualified and unqualified imports work the same way (they are both indirections).
- You can split code into multiple modules and it will continue to work (as long as you don’t try to change the values of imports).

16.3.6 Support for cyclic dependencies

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, then? Occasionally, you can’t get around them, which is why support for them is an important feature. A later section has more information.

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

16.3.6.1 Cyclic dependencies in CommonJS

The following CommonJS code correctly handles two modules a and b cyclically depending on each other.

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

//------ b.js ------
var a = require('a'); // (i)
function bar() {
    if (Math.random()) {
        a.foo(); // (ii)
    }
}
exports.bar = bar;
```

If module a is imported first then, in line i, module b gets a’s exports object before the exports are added to it. Therefore, b cannot access a.foo in its top level, but that property exists once the execution of a is finished. If bar() is called afterwards then the method call in line ii works.

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. There, you export single values instead of objects:

  ```
  module.exports = function () { ··· };
  ```

  If module a did that then module b's variable a would not be updated once the assignment is made. It would continue to refer to the original exports object.

- You can't use named exports directly. That is, module b can't import 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 a.foo.

These limitations mean that both exporter and importers must be aware of cyclic dependencies and support them explicitly.

16.3.6.2 Cyclic dependencies in ECMAScript 6

ES6 modules support cyclic dependencies automatically. That is, they do not have the two limitations of CommonJS modules that were mentioned in the previous section: default exports work, as do unqualified named imports (lines i and iii in the following example). Therefore, you can implement modules that cyclically depend on each other as follows.

```
//------ a.js ------
import {bar} from 'b'; // (i)
export function foo() {
    bar(); // (ii)
}

//------ b.js ------
import {foo} from 'a'; // (iii)
export function bar() {
    if (Math.random()) {
        foo(); // (iv)
    }
}
```

This code works, because, as explained in the previous section, imports are views on exports. That means that even unqualified imports (such as bar in line ii and foo in line iv) are indirections that refer to the original data. Thus, in the face of cyclic dependencies, it doesn't matter whether you access a named export via an unqualified import or via its module: There is an indirection involved in either case and it always works.

16.4 Importing and exporting in detail

16.4.1 Importing styles

ECMAScript 6 provides several styles of importing[2]:

- Default import:

  ```
  import localName from 'src/my_lib';
  ```

- Namespace import: imports the module as an object (with one property per named export).

  ```
  import * as my_lib from 'src/my_lib';
  ```

- Named imports:

  ```
  import { name1, name2 } from 'src/my_lib';
  ```

  You can rename named imports:

```
// Renaming: import `name1` as `localName1`
import { name1 as localName1, name2 } from 'src/my_lib';

// Renaming: import the default export as `foo`
import { default as foo } from 'src/my_lib';
```

- Empty import: only loads the module, doesn't import anything. The first such import in a program executes the body of the module.

```
import 'src/my_lib';
```

There are only two ways to combine these styles and the order in which they appear is fixed; the default export always comes first.

- Combining a default import with a namespace import:

```
import theDefault, * as my_lib from 'src/my_lib';
```

- Combining a default import with named imports

```
import theDefault, { name1, name2 } from 'src/my_lib';
```

16.4.2 Named exporting styles: inline versus clause

There are two ways in which you can export named things inside modules.

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 {
   ···
}
```

On the other hand, you can list everything you want to export at the end of the module (which is 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 FOO, myFunc };
```

16.4.3 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';
```

Default exports are ignored[3] by `export *`.

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

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

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

16.4.3.1 Making a re-export the default export

The following statement makes the default export of another module`foo` the default export of the current module:

```
export { default } from 'foo';
```

The following statement makes the named export `myFunc` of module `foo` the default export of the current module:

```
export { myFunc as default } from 'foo';
```

16.4.4 All exporting styles

ECMAScript 6 provides several styles of exporting[4]:

- Re-exporting:
  - Re-export everything (except for the default export):

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

  - Re-export via a clause:

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

    export { default } from 'src/other_module';
    export { default as foo } from 'src/other_module';
    export { foo as default } from 'src/other_module';
    ```

- Named exporting via a clause:

  ```
  export { MY_CONST as FOO, myFunc };
  export { foo as default };
  ```

- Inline named exports:
  - Variable declarations:

    ```
    export var foo;
    export let foo;
    export const foo;
    ```

  - Function declarations:

    ```
    export function myFunc() {}
    export function* myGenFunc() {}
    ```

  - Class declarations:

    ```
    export class MyClass {}
    ```

- Default export:
  - Function declarations (can be anonymous here):

    ```
    export default function myFunc() {}
    export default function () {}

    export default function* myGenFunc() {}
    export default function* () {}
    ```

  - Class declarations (can be anonymous here):

    ```
    export default class MyClass {}
    export default class {}
    ```

  - Expressions: export values. Note the semicolons at the end.

    ```
    export default foo;
    export default 'Hello world!';
    export default 3 * 7;
    export default (function () {});
    ```

16.4.5 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.

16.4.5.1 Recommendation: avoid mixing default exports and named exports

I generally recommend to keep the two kinds of exporting separate: per module, either only have a default export or only have named exports.

However, that is not a very strong recommendation; it occasionally may make sense to mix the two kinds. One example is a module that default-exports an entity. For unit tests, one could additionally make some of the internals available via named exports.

16.4.5.2 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 function foo() {} // function declaration!

//------ module2.js ------
function foo() {}
export { foo as default };
```

16.4.5.3 default: OK as export name, but not as variable name

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.

That means that default can only appear on the left-hand side of a renaming import:

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

And it can only appear on the right-hand side of a renaming export:

```
export { foo as default };
```

In re-exporting, both sides of the as are export names:

```
export { myFunc as default } from 'foo';
export { default as otherFunc } from 'foo';
```

```
// The following two statements are equivalent:
export { default } from 'foo';
export { default as default } from 'foo';
```

16.5 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
- Configure module loading

The module loader API is not part of the ES6 standard

It will be specified in a separate document, the “JavaScript Loader Standard”, that will be evolved more dynamically than the language specification. The repository for that document states:

> [The JavaScript Loader Standard] consolidates work on the ECMAScript module loading semantics with the integration points of Web browsers, as well as Node.js.

The module loader API is work in progress

As you can see in the repository of the JavaScript Loader Standard, the module loader API is still work in progress. Everything you read about it in this book is tentative. To get an impression of what the API may look like, you can take a look at the ES6 Module Loader Polyfill on GitHub.

16.5.1 Loaders

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 default instance in the global variable System (the *system loader*), which implements its specific style of module loading.

16.5.2 Loader method: importing modules

You can programmatically import a module, via an API based on 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 the section on modules versus scripts 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
});
```

16.5.3 More loader methods

Loaders have more methods. Three important ones are:

- 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.

16.5.4 Configuring module loading

The module loader API will have various hooks for configuring the loading process. Use cases include:

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.

16.6 Using ES6 modules in browsers

Let’s look at how ES6 modules are supported in browsers.

Support for ES6 modules in browsers is work in progress

Similarly to module loading, other aspects of support for modules in browsers are still being worked on. Everything you read here may change.

16.6.1 Browsers: asynchronous modules versus synchronous scripts

In browsers, there are two different kinds of entities: scripts and modules. They have slightly different syntax and work differently.

This is an overview of the differences, details are explained later:

	Scripts	**Modules**
HTML element	`<script>`	`<script type="module">`
Default mode	non-strict	strict
Top-level variables are	global	local to module
Value of this at top level	window	undefined
Executed	synchronously	asynchronously
Declarative imports (import statement)	no	yes
Programmatic imports (Promise-based API)	yes	yes
File extension	.js	.js

16.6.1.1 Scripts

Scripts are the traditional browser way to embed JavaScript and to refer to external JavaScript files. Scripts have an internet media type that is used as:

- The content type of JavaScript files delivered via a web server.
- The value of the attribute type of `<script>` elements. Note that for HTML5, the recommendation is to omit the type attribute in `<script>` elements if they contain or refer to JavaScript.

The following are the most important values:

- text/javascript: is a legacy value and used as the default if you omit the type attribute in a script tag. It is the safest choice for Internet Explorer 8 and earlier.
- application/javascript: is recommended for current browsers.

Scripts are normally loaded or executed synchronously. The JavaScript thread stops until the code has been loaded or executed.

16.6.1.2 Modules

To be in line with JavaScript’s usual run-to-completion semantics, the body of a module must be executed without interruption. That leaves two options for importing modules:

1. Load modules synchronously, while the body is executed. That is what Node.js does.
2. Load all modules asynchronously, before the body is executed. That is how AMD modules are handled. It is the best option for browsers, because modules are loaded over the internet and execution doesn't have to pause while they are. As an added benefit, this approach allows one to load multiple modules in parallel.

ECMAScript 6 gives you the best of both worlds: The synchronous syntax of Node.js plus the asynchronous loading of AMD. To make both possible, ES6 modules are syntactically less flexible than Node.js modules: Imports and exports must happen at the top level. That means that they can't be conditional, either. This restriction allows an ES6 module loader to analyze statically what modules are imported by a module and load them before executing its body.

The synchronous nature of scripts prevents them from becoming modules. Scripts cannot even import modules declaratively (you have to use the programmatic module loader API if you want to do so).

Modules can be used from browsers via a new variant of the`<script>` element that is completely asynchronous:

```
<script type="module">
   import $ from 'lib/jquery';
   var x = 123;

   // The current scope is not global
   console.log('$' in window); // false
   console.log('x' in window); // false

   // `this` is undefined
   console.log(this === undefined); // true
</script>
```

As you can see, the element has its own scope and variables "inside" it are local to that scope. Note that module code is implicitly in strict mode. This is great news – no more `'use strict'`.

Similar to normal `<script>` elements, `<script type="module">` can also be used to load external modules. For example, the following tag starts a web application via a `main` module (the attribute name `import` is my invention, it isn't yet clear what name will be used).

```
<script type="module" import="impl/main"></script>
```

The advantage of supporting modules in HTML via a custom`<script>` type is that it is easy to bring that support to older engines via a polyfill (a library). There may or may not eventually be a dedicated element for modules (e.g. `<module>`).

16.6.1.3 Module or script – a matter of context

Whether a file is a module or a script is only determined by how it is imported or loaded. Most modules have either imports or exports and can thus be detected. But if a module has neither then it is indistinguishable from a script. For example:

```
var x = 123;
```

The semantics of this piece of code differs depending on whether it is interpreted as a module or as a script:

- As a module, the variable `x` is created in module scope.
- As a script, the variable `x` becomes a global variable and a property of the global object (`window` in browsers).

More realistic example is a module that installs something, e.g. a polyfill in global variables or a global event listener. Such a module neither imports nor exports anything and is activated via an empty import:

```
import './my_module';
```

Sources of this section

- “Modules: Status Update”, slides by David Herman.
- “Modules vs Scripts”, an email by David Herman.

16.7 Details: imports as views on exports

The code in this section is available on GitHub.

Imports work differently in CommonJS and ES6:

- In CommonJS, imports are copies of exported values.
- In ES6, imports are live read-only views on exported values.

The following sections explain what that means.

16.7.1 In CommonJS, imports are copies of exported values

With CommonJS (Node.js) modules, things work in relatively familiar ways.

If you import a value into a variable, the value is copied twice: once when it is exported (line A) and once it is imported (line B).

```
//------ lib.js ------
var counter = 3;
function incCounter() {
   counter++;
}
module.exports = {
   counter: counter, // (A)
   incCounter: incCounter,
};

//------ main1.js ------
var counter = require('./lib').counter; // (B)
var incCounter = require('./lib').incCounter;

// The imported value is a (disconnected) copy of a copy
console.log(counter); // 3
incCounter();
console.log(counter); // 3

// The imported value can be changed
counter++;
console.log(counter); // 4
```

If you access the value via the exports object, it is still copied once, on export:

```
//------ main2.js ------
var lib = require('./lib');

// The imported value is a (disconnected) copy
console.log(lib.counter); // 3
lib.incCounter();
console.log(lib.counter); // 3

// The imported value can be changed
lib.counter++;
console.log(lib.counter); // 4
```

16.7.2 In ES6, imports are live read-only views on exported values

In contrast to CommonJS, imports are views on exported values. In other words, every import is a live connection to the exported data. Imports are read-only:

- Unqualified imports (`import x from 'foo'`) are like `const`-declared variables.
- The properties of a module object `foo` (`import * as foo from 'foo'`) are like the properties of a frozen object.

The following code demonstrates how imports are like views:

```
//------ lib.js ------
export let counter = 3;
export function incCounter() {
   counter++;
}

//------ main1.js ------
import { counter, incCounter } from './lib';
```

```
// The imported value `counter` is live
console.log(counter); // 3
incCounter();
console.log(counter); // 4

// The imported value can't be changed
counter++; // TypeError
```

(Ad, please don't block.)

See how your visitors are really using your website.
ads via Carbon

```
import * as lib from './lib';

// The imported value `counter` is live
console.log(lib.counter); // 3
lib.incCounter();
console.log(lib.counter); // 4

// The imported value can't be changed
lib.counter++; // TypeError
```

Note that while you can't change the values of imports, you can change the objects that they are referring to. For example:

```
//------ lib.js ------
export let obj = {};

//------ main.js ------
import { obj } from './lib';

obj.prop = 123; // OK
obj = {}; // TypeError
```

16.7.2.1 Why a new approach to importing?

Why introduce such a relatively complicated mechanism for importing that deviates from established practices?

- Cyclic dependencies: The main advantage is that it supports cyclic dependencies even for unqualified imports.
- Qualified and unqualified imports work the same. In CommonJS, they don't: a qualified import provides direct access to a property of a module's export object, an unqualified import is a copy of it.
- You can split code into multiple modules and it will continue to work (as long as you don't try to change the values of imports).
- On the flip side, *module folding*, combining multiple modules into a single module becomes simpler, too.

In my experience, ES6 imports just work, you rarely have to think about what's going on under the hood.

16.7.3 Implementing views

How do imports work as views of exports under the hood? Exports are managed via the data structure *export entry*. All export entries (except those for re-exports) have the following two names:

- Local name: is the name under which the export is stored inside the module.
- Export name: is the name that importing modules need to use to access the export.

After you have imported an entity, that entity is always accessed via a pointer that has the two components *module* and *local name*. In other words, that pointer refers to a *binding* (the storage space of a variable) inside a module.

Let's examine the export names and local names created by various kinds of exporting. The following table (adapted from the ES6 spec) gives an overview, subsequent sections have more details.

Statement	Local name	Export name
export {v};	'v'	'v'
export {v as x};	'v'	'x'
export const v = 123;	'v'	'v'
export function f() {}	'f'	'f'

Statement	Local name	Export name
export default function f() {}	'f'	'default'
export default function () {}	'*default*'	'default'
export default 123;	'*default*'	'default'

16.7.3.1 Export clause

```
function foo() {}
export { foo };
```

- Local name: foo
- Export name: foo

```
function foo() {}
export { foo as bar };
```

- Local name: foo
- Export name: bar

16.7.3.2 Inline exports

This is an inline export:

```
export function foo() {}
```

It is equivalent to the following code:

```
function foo() {}
export { foo };
```

Therefore, we have the following names:

- Local name: foo
- Export name: foo

16.7.3.3 Default exports

There are two kinds of default exports:

- Default exports of *hoistable declarations* (function declarations, generator function declarations) and class declarations are similar to normal inline exports in that named local entities are created and tagged.
- All other default exports are about exporting the results of expressions.

16.7.3.3.1 Default-exporting expressions

The following code default-exports the result of the expression 123:

```
export default 123;
```

It is equivalent to:

```
const *default* = 123; // *not* legal JavaScript
export { *default* as default };
```

If you default-export an expression, you get:

- Local name: *default*
- Export name: default

The local name was chosen so that it wouldn't clash with any other local name.

Note that a default export still leads to a binding being created. But, due to *default* not being a legal identifier, you can't access that binding from inside the module.

16.7.3.3.2 Default-exporting hoistable declarations and class declarations

The following code default-exports a function declaration:

```
export default function foo() {}
```

It is equivalent to:

```
function foo() {}
```

```
export { foo as default };
```

The names are:

- Local name: foo
- Export name: default

That means that you can change the value of the default export from within the module, by assigning a different value to foo.

(Only) for default exports, you can also omit the name of a function declaration:

```
export default function () {}
```

That is equivalent to:

```
function *default*() {} // *not* legal JavaScript
export { *default* as default };
```

The names are:

- Local name: *default*
- Export name: default

Default-exporting generator declarations and class declarations works similarly to default-exporting function declarations.

16.7.4 Imports as views in the spec

This section gives pointers into the ECMAScript 2015 (ES6) language specification.

Managing imports:

- CreateImportBinding() creates local bindings for imports.
- GetBindingValue() is used to access them.
- ModuleDeclarationInstantiation() sets up the environment of a module (compare: FunctionDeclarationInstantiation(), BlockDeclarationInstantiation()).

The export names and local names created by the various kinds of exports are shown in table 42 in the section "Source Text Module Records". The section "Static Semantics: ExportEntries" has more details. You can see that export entries are set up statically (before evaluating the module), evaluating export statements is described in the section "Runtime Semantics: Evaluation".

16.8 Design goals for ES6 modules

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.

16.8.1 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.

16.8.2 Static module structure

Current JavaScript module formats have a dynamic structure: What is imported and exported can change at runtime. One reason why ES6 introduced its own module format is to enable a static structure, which has several benefits. But before we go into those, let's examine what the structure being static means.

It means that you can determine imports and exports at compile time (statically) – you only need to look at the source code, you don't have to execute it. ES6 enforces this syntactically: You can only import and export at the top level (never nested inside a conditional statement). And import and export statements have no dynamic parts (no variables etc. are allowed).

The following are two examples of CommonJS modules that don't have a static structure. In the first example, you have to run the code to find out what it imports:

```
var my_lib;
if (Math.random()) {
   my_lib = require('foo');
} else {
   my_lib = 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 modules are less flexible and force you to be static. As a result, you get several benefits, which are described next.

16.8.2.1 Benefit: dead code elimination during bundling

In frontend development, modules are usually handled as follows:

- During development, code exists as many, often small, modules.
- For deployment, these modules are *bundled* into a few, relatively large, files.

The reasons for bundling are:

1. Fewer files need to be retrieved in order to load all modules.
2. Compressing the bundled file is slightly more efficient than compressing separate files.
3. During bundling, unused exports can be removed, potentially resulting in significant space savings.

Reason #1 is important for HTTP/1, where the cost for requesting a file is relatively high. That will change with HTTP/2, which is why this reason doesn't matter there.

Reason #3 will remain compelling. It can only be achieved with a module format that has a static structure.

16.8.2.2 Benefit: compact bundling, no custom bundle format

The module bundler Rollup proved that ES6 modules can be combined efficiently, because they all fit into a single scope (after renaming variables to eliminate name clashes). This is possible due to two characteristics of ES6 modules:

- Their static structure means that the bundle format does not have to account for conditionally loaded modules (a common technique for doing so is putting module code in functions).
- Imports being read-only views on exports means that you don't have to copy exports, you can refer to them directly.

As an example, consider the following two ES6 modules.

```
// lib.js
export function foo() {}
export function bar() {}

// main.js
import {foo} from './lib.js';
console.log(foo());
```

Rollup can bundle these two ES6 modules into the following single ES6 module (note the eliminated unused export `bar`):

```
function foo() {}
```

```
console.log(foo());
```

Another benefit of Rollup's approach is that the bundle does not have a custom format, it is just an ES6 module.

16.8.2.3 Benefit: faster lookup of imports

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
```

16.8.2.4 Benefit: 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.

16.8.2.5 Benefit: 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.

16.8.2.6 Benefit: 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).

16.8.2.7 Benefit: 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.

16.8.2.8 Source of this section

- "Static module resolution" by David Herman

16.8.3 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).

16.8.4 Support for cyclic dependencies between modules

Support for cyclic dependencies was a key goal for ES6 modules. Here is why:

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 the system doesn't break while you are refactoring.

The Node.js documentation acknowledges the importance of cyclic dependencies 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.

16.9 FAQ: modules

16.9.1 Can I use a variable to specify from which module I want to import?

The `import` statement is completely static: its module specifier is always fixed. If you want to dynamically determine what module to load, you need to use the programmatic loader API:

```
const moduleSpecifier = 'module_' + Math.random();
System.import(moduleSpecifier)
.then(the_module => {
    // Use the_module
})
```

16.9.2 Can I import a module conditionally or on demand?

Import statements must always be at the top level of modules. That means that you can’t nest them inside `if` statements, functions, etc. Therefore, you have to use the programmatic loader API if you want to load a module conditionally or on demand:

```
if (Math.random()) {
    System.import('some_module')
    .then(some_module => {
        // Use some_module
    })
}
```

16.9.3 Can I use variables in an `import` statement?

No, you can’t. Remember – what is imported must not depend on anything that is computed at runtime. Therefore:

```
// Illegal syntax:
import foo from 'some_module'+SUFFIX;
```

16.9.4 Can I use destructuring in an `import` statement?

No you can’t. The `import` statement only looks like destructuring, but is completely different (static, imports are views, etc.).

Therefore, you can’t do something like this in ES6:

```
// Illegal syntax:
import { foo: { bar } } from 'some_module';
```

16.9.5 Are named exports necessary? Why not default-export objects?

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

16.9.6 Can I `eval()` the code of module?

No, you can’t. Modules are too high-level a construct for `eval()`. The module loader API provides the means for creating modules from strings. Syntactically, `eval()` accepts scripts (which don’t allow `import` and `export`), not modules.

16.10 Advantages 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 several new features:

- More compact syntax
- Static module structure (helping with dead code elimination, optimizations, static checking and more)
- Automatic support for cyclic dependencies

ES6 modules 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.

16.11 Further reading

- **CommonJS versus 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.

Next: IV Collections