A Comprehensive Look at ES6 Modules

**Summary**: in this tutorial, you will learn about **ES6 modules** and how to export variables, functions, and classes from a module, and reuse them in other modules.

An ES6 module is a JavaScript file that executes in strict mode only. It means that any [variables](https://www.javascripttutorial.net/es6/javascript-let/) or [functions](https://www.javascripttutorial.net/javascript-function/) declared in the module won’t be added automatically to the global scope.

Executing modules on web browsers

First, create a new file called message.js and add the following code:

export let message = 'ES6 Modules';

Code language: JavaScript (javascript)

The message.js is a module in ES6 that contains the message variable. The export statement exposes the message variable to other modules.

Second, create another new file named app.js that uses the message.js module. The app.js module creates a new heading 1 (h1) element and attaches it to an HTML page. The import statement imports the message variable from the message.js module.

import { message } from './message.js'

const h1 = document.createElement('h1');

h1.textContent = message

document.body.appendChild(h1)

Code language: JavaScript (javascript)

Third, create a new HTML page that uses the app.js module:

<!DOCTYPE html>

**<html>**

**<head>**

**<meta charset="utf-8">**

**<title>**ES6 Modules**</title>**

**</head>**

**<body>**

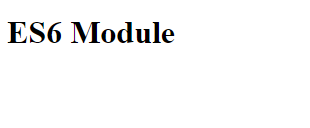
**<script type="module" src="./app.js"></script>**

**</body>**

**</html>**

Code language: HTML, XML (xml)

Note that we used the type="module" in the <script> tag to load the app.js module. If you view the page on a web browser, you will see the following page:



Let’s examine the export and import statements in more detail.

Exporting

To export a [variable](https://www.javascripttutorial.net/javascript-variables/), a [function](https://www.javascripttutorial.net/javascript-function/), or a [class](https://www.javascripttutorial.net/es6/javascript-class/), you place the export keyword in front of it as follows:

*// log.js*

export let message = 'Hi';

export function getMessage() {

return message;

}

export function setMessage(msg) {

message = msg;

}

export class Logger {

}

Code language: JavaScript (javascript)

In this example, we have the log.js module with a variable, two functions, and one class. We used the export keyword to exports all identifiers in the module.

Note that the export keyword requires the function or class to have a name to be exported. You can’t export an anonymous function or class using this syntax.

JavaScript allows you to define a variable, a function, or a class first and then export it later as follows:

*// foo.js*

function foo() {

console.log('foo');

}

function bar() {

console.log('bar');

}

export foo;

Code language: JavaScript (javascript)

In this example, we defined the foo() function first and then exported it. Since we didn’t export the bar() function, we couldn’t access it in other modules. The bar() function is inaccessible outside the module or we say it is private.

Importing

Once you define a module with exports, you can access the exported variables, functions, and classes in another module by using the import keyword. The following illustrates the syntax:

import { what, ever } from './other\_module.js';

Code language: JavaScript (javascript)

In this syntax:

* First, specify what to import inside the curly braces, which are called bindings.
* Then, specify the module from which you import the given bindings.

Note that when you import a binding from a module, the binding behaves like it was defined using [const](https://www.javascripttutorial.net/es6/javascript-const/). It means you can’t have another identifier with the same name or change the value of the binding.

See the following example:

*// greeting.js*

export let message = 'Hi';

export function setMessage(msg) {

message = msg;

}

Code language: JavaScript (javascript)

When you import the message variable and setMessage() function, you can use the setMessage() function to change the value of the message variable as shown below:

*// app.js*

import {message, setMessage } from './greeting.js';

console.log(message); *// 'Hi'*

setMessage('Hello');

console.log(message); *// 'Hello'*

Code language: JavaScript (javascript)

However, you can’t change the value of the message variable directly. The following expression causes an error:

message = 'Hallo'; *// error*

Code language: JavaScript (javascript)

Behind the scenes, when you called the setMessage() function. JavaScript went back to the greeting.js module and executed the code in there and changed the message variable. The change was then automatically reflected on the imported message binding.

The message binding in the app.js is the local name for exported message identifier. So basically the message variables in the app.js and greeting.js modules aren’t the same.

Import a single binding

Suppose you have a module with the foo variable as follows:

*// foo.js*

export foo = 10;

Code language: JavaScript (javascript)

Then, in another module, you can reuse the foo variable:

*// app.js*

import { foo } from './foo.js';

console.log(foo); *// 10;*

Code language: JavaScript (javascript)

However, you can’t change the value of foo. If you attempt to do so, you will get an error:

foo = 20; *// throws an error*

Code language: JavaScript (javascript)

Import multiple bindings

Suppose you have the cal.js module as follows:

*// cal.js*

export let a = 10,

b = 20,

result = 0;

export function sum() {

result = a + b;

return result;

}

export function multiply() {

result = a \* b;

return result;

}

Code language: JavaScript (javascript)

And you want to import these bindings from the cal.js, you can explicitly list them as follows:

import {a, b, result, sum, multiply } from './cal.js';

sum();

console.log(result); *// 30*

multiply();

console.log(result); *// 200*

Code language: JavaScript (javascript)

Import an entire module as an object

To import everything from a module as a single object, you use the asterisk (\*) pattern as follows:

import \* as cal from './cal.js';

Code language: JavaScript (javascript)

In this example, we imported all bindings from the cal.js module as the cal object. In this case, all the bindings become properties of the cal object, so you can access them as shown below:

cal.a;

cal.b;

cal.sum();

Code language: CSS (css)

This import is called *namespace import*.

It’s important to keep in mind that the imported module executes *only once* even if import multiple times. Consider this example:

import { a } from './cal.js';

import { b } from './cal.js';

import {result} from './cal.js';

Code language: JavaScript (javascript)

After the first import statement, the cal.js module is executed and loaded into the memory, and it is reused whenever it is referenced by the subsequent import statement.

Limitation of import and export statements

Note that you must use the import or export statement *outside* other statements and functions. The following example causes a SyntaxError:

if( requiredSum ) {

export sum;

}

Code language: JavaScript (javascript)

Because we used the export statement inside the [if](https://www.javascripttutorial.net/javascript-if-else/) statement. Similarly, the following import statement also causes a SyntaxError:

function importSum() {

import {sum} from './cal.js';

}

Code language: JavaScript (javascript)

Because we used the import statement inside a function.

The reason for the error is that JavaScript must *statically* determine what will be exported and imported.

Note that ES2020 introduced the function-like object [import()](https://www.javascripttutorial.net/es-next/javascript-import/) that allows you to dynamically import a module.

Aliasing

JavaScript allows you to create aliases for variables, functions, or classes when you export and import. See the following math.js module:

*// math.js*

function add( a, b ) {

return a + b;

}

export { add as sum };

Code language: JavaScript (javascript)

In this example, instead of exporting the add() function, we used the as keyword to assign the sum() function an alias.

So when you import the add() function from the math.js module, you must use sum instead:

import { sum } from './math.js';

Code language: JavaScript (javascript)

If you want to use a different name when you import, you can use the as keyword as follows:

import {sum as total} from './math.js';

Code language: JavaScript (javascript)

Re-exporting a binding

It’s possible to export bindings that you have imported. This is called re-exporting. For example:

import { sum } from './math.js';

export { sum };

Code language: JavaScript (javascript)

In this example, we imported sum from the math.js module and re-export it. The following statement is equivalent to the statements above:

export {sum} from './math.js';

Code language: JavaScript (javascript)

In case you want to rename the bindings before re-exporting, you use the as keyword. The following example of imports sum from the math.js module and re-export it as add.

export { sum as add } from './math.js';

Code language: JavaScript (javascript)

If you want to export all the bindings from another module, you can use the asterisk (\*):

export \* from './cal.js';

Code language: JavaScript (javascript)

Importing without bindings

Sometimes, you want to develop a module that doesn’t export anything, for example, you may want to add a new method to a built-in object such as the [Array](https://www.javascripttutorial.net/javascript-array/).

*// array.js*

if (!Array.prototype.contain) {

Array.prototype.contain = function(e) {

*// contain implementation*

*// ...*

}

}

Code language: JavaScript (javascript)

Now, you can import the module without any binding and use the contain() method defined in the array.js module as follows:

import './array.js';

[1,2,3].contain(2); *// true*

Code language: JavaScript (javascript)

Default exports

A module can have one and only one default export. The default export is easier to import. The default for a module can be a variable, a function, or a class.

The following is the sort.js module with a default export.

*// sort.js*

export default function(arr) {

*// sorting here*

}

Code language: JavaScript (javascript)

Note that you don’t need to specify the name for the function because the module represents the function name.

import sort from sort.js;

sort([2,1,3]);

Code language: JavaScript (javascript)

As you see, the  sort identifier represents the default function of the sort.js module. Notice that we didn’t use the curly brace {} surrounding the  sort identifier.

Let’s change the sort.js module to include the default export as well as the non-default one:

*// sort.js*

export default function(arr) {

*// sorting here*

}

export function heapSort(arr) {

*// heapsort*

}

Code language: JavaScript (javascript)

To import both default and non-default bindings, you specify a list of bindings after the import keyword with the following rules:

* The default binding must come first.
* The non-default binding must be surrounded by curly braces.

See the following example:

import sort, {heapSort} from './sort.js';

sort([2,1,3]);

heapSort([3,1,2]);

Code language: JavaScript (javascript)

To rename the default export, you also use the as keyword as follows:

import { default as quicksort, heapSort} from './sort.js';

Code language: JavaScript (javascript)

In this tutorial, you have learned about ES6 modules and how to export bindings from a module and import them into another module.

JavaScript import

**Summary**: in this tutorial, you’ll learn how to dynamically import modules by using the function-like import() in ES2020.

Introduction to the JavaScript import()

ES6 introduced the module concept that allows you to develop modular JavaScript code. Suppose you have the following simple HTML document that has one button:

<!DOCTYPE html>

**<html>**

**<head>**

**<title>**Module Dynamic Import**</title>**

**</head>**

**<body>**

**<button id="show">**Show Dialog**</button>**

**<script type="module" src="js/app.js"></script>**

**</body>**

**</html>**

Code language: JavaScript (javascript)

When users click the button, you want to show a dialog. To make the code more organized, you develop a module called dialog.js:

export function show(message) {

alert(message);

}

Code language: JavaScript (javascript)

And use the show() function in the app.js:

import {show} from './dialog.js';

let btn = document.querySelector('#show');

btn.addEventListener('click', function () {

show('Hi');

});

Code language: JavaScript (javascript)

Prior to ES2020, it was not possible to dynamically load the dialog.js module when needed. The following will cause an error:

let btn = document.querySelector('#show');

btn.addEventListener('click', function () {

import {show} from './dialog.js';

show('Hi');

});

Code language: JavaScript (javascript)

The above code attempts to load the dialog.js module only when the button is clicked.

ES2020 introduced the dynamic import of the module via the function-like import() with the following syntax:

import(moduleSpecifier);

Code language: JavaScript (javascript)

The import() allows you to dynamically import a module when needed. Here is how the import() works:

* The import() accepts a module specifier (moduleSpecifier) that has the same format as the module specifier used for the import statement. In addition, the moduleSpecifier can be an expression that evaluates to a string.
* The import() returns a [Promise](https://www.javascripttutorial.net/es6/javascript-promises/) that will be fulfilled once the module is loaded completely.

To load the dialog.js dynamically, you can use the import() as follows:

let btn = document.querySelector('#show');

btn.addEventListener('click', function() {

import('./dialog.js')

.then(( dialog ) => {

dialog.show();

})

.catch( error => {

*// handle error here*

});

});

Code language: JavaScript (javascript)

Since the import() returns a [Promise](https://www.javascripttutorial.net/es6/javascript-promises/), you can use the [async/await](https://www.javascripttutorial.net/es-next/javascript-async-await/) in the app.js module like this:

let btn = document.querySelector('#show');

btn.addEventListener('click', function () {

(async () => {

try {

let dialog = await import('./dialog.js');

dialog.show('Hi')

} catch (error) {

console.log(error);

}

})();

});

Code language: JavaScript (javascript)

Some practical use cases of JavaScript import()

The import() has the following practical use cases:

1) Loading module on demand

Some functionality may not need to be available when applications start. To decrease the loading time, you can place such functionality in modules and use the import() to load them on demand like this:

function eventHandler() {

import('./module1.js')

.then((ns) => {

*// use the module*

ns.func();

})

.catch((error) => {

*// handle error*

});

}

Code language: JavaScript (javascript)

2) Loading modules based on conditions

When placing the import() inside the conditional statement such as [if-else](https://www.javascripttutorial.net/javascript-if-else/), you can load modules based on a specific condition. The following example loads a module that targets a specific platform:

if( isSpecificPlatform() ) {

import('./platform.js')

.then((ns) => {

ns=>f();

});

}

Code language: JavaScript (javascript)

3) Computed module specifiers

The module specifier is an expression that allows you to decide which module to load at runtime.

For example, you can load a module based on the user’s locale to show the message in the user’s specific language:

let lang = `message\_${getUserLocale()}.js`;

import(lang)

.then(...);

Code language: JavaScript (javascript)

More on the JavaScript import()

Using object destructuring

If a module has multiple exports, you can use the [object destructuring](https://www.javascripttutorial.net/es6/javascript-object-destructuring/) to receive the exporting objects. Suppose the dialog.js has two functions:

export function show(message) {

alert(message);

}

export function hide(message) {

console.log('Hide it...');

}

Code language: JavaScript (javascript)

In the app.js, you can use the object destructuring as follows:

let btn = document.querySelector('#show');

btn.addEventListener('click', function () {

(async () => {

try {

*// use object destructuring*

let {

show,

hide

} = await import('./dialog.js');

*// use the functions*

show('Hi');

hide();

} catch (err) {

console.log(err);

}

})();

});

Code language: JavaScript (javascript)

Dynamically loading multiple modules

To load multiple modules dynamically, you can use the [Promise.all()](https://www.javascripttutorial.net/es6/javascript-promise-all/) method:

Promise.all([

import(module1),

import(module2),

...])

.then(([module1,module2,module3]) => {

*// use the modules*

});

Code language: JavaScript (javascript)

Accessing the default export

If a module has a default export, you can access it using the default keyword. For example:

import(moduleSpecifier)

.then((module) => {

*// access the default export*

console.log(module.default);

});

Code language: JavaScript (javascript)

Summary

* Use JavaScript import() to dynamically load a module. The import() returns a Promise that will be fulfilled once the module is loaded completely.
* Use the async / await to handle the result of the import().
* Use the Promise.all() method to load multiple modules at once.
* Use the object destructuring to assign variables to the exporting objects of a module.
* Use the default keyword to access the default export.

JavaScript Top-level await

**Summary**: in this tutorial, you’ll learn about the JavaScript top-level await and its use cases.

Introduction to the JavaScript top-level await

ES2020 introduced the top-level await feature that allows a module to behave like an async function. A [module](https://www.javascripttutorial.net/es6/es6-modules/) that imports the top-level await module will wait for it to load before evaluating its body.

To better understand the top-level await feature, we’ll take an example:

In this example, we’ll have three files: index.html, app.mjs, and user.mjs:

* The index.html uses the app.mjs file.
* The app.mjs imports the user.mjs file.
* The user.mjs fetches the user data in JSON format from an API with the URL endpoint <https://jsonplaceholder.typicode.com/users>

Here’s the index file that uses the app.mjs module:

<!DOCTYPE html>

**<html lang="en">**

**<head>**

**<meta charset="UTF-8">**

**<meta name="viewport" content="width=device-width, initial-scale=1.0">**

**<title>**JavaScript Top-Level Await Demo**</title>**

**</head>**

**<body>**

**<div class="container"></div>**

**<script type="module" src="app.mjs"></script>**

**</body>**

**</html>**

Code language: HTML, XML (xml)

The following shows the user.mjs file:

let users;

(async () => {

const url = 'https://jsonplaceholder.typicode.com/users';

const response = await fetch(url);

users = await response.json();

})();

export { users };

Code language: JavaScript (javascript)

The user.mjs module uses the [fetch API](https://www.javascripttutorial.net/javascript-fetch-api/) to get the users in JSON format from an API and export it.

Because we can only use the await keyword inside an async function (before ES2020), we need to wrap the API call inside an immediately invoked async function expression (IIAFE).

The following shows the app.mjs module:

import { users } from './user.mjs';

function render(users) {

if (!users) {

throw 'The user list is not available';

}

const list = users

.map((user) => {

return `<li> ${user.name}(<a href="email:${user.email}">${user.email}</a>)</li>`;

})

.join('');

return `<ol>${list}</ol>`;

}

const container = document.querySelector('.container');

try {

container.innerHTML = render(users);

} catch (e) {

container.innerHTML = e;

}

Code language: JavaScript (javascript)

How it works.

First, import users from the user.mjs module:

import { users } from './user.mjs';

Code language: JavaScript (javascript)

Second, create a render() function that renders the user list to an ordered list in HTML format:

function render(users) {

if (!users) {

throw 'The user list is not available.';

}

const list = users

.map((user) => {

return `<li> ${user.name}(<a href="email:${user.email}">${user.email}</a>)</li>`;

})

.join('');

return `<ol>${list}</ol>`;

}

Code language: JavaScript (javascript)

Third, add the user list to the HTML element with the class .container:

const container = document.querySelector('.container');

try {

container.innerHTML = render(users);

} catch (e) {

container.innerHTML = e;

}

Code language: JavaScript (javascript)

If you open the index.html, you’ll see the following message:

The user list is not available.

Code language: PHP (php)

The following shows the main flow:

In this flow:

* First, the app.mjs imports the user.mjs module.
* Second, the user.mjs module executes and make an API call.
* Third, while the second step is still on-going, the app.mjs starts using the users data imported from the user.mjs module.

Since the step 2 has not completed, the users variable was undefined. Therefore, you saw the error message on the page.

Workaround

To fix the issue, you can export a Promise from the user.mjs module and wait for the API call to complete before using its result.

The following shows the new version of the user.mjs module:

let users;

export default (async () => {

const url = 'https://jsonplaceholder.typicode.com/users';

const response = await fetch(url);

users = await response.json();

})();

export { users };

Code language: JavaScript (javascript)

In this new version, the user.mjs model exports the users and a Promise as a default export.

In the app.mjs imports the promise and users from the user.mjs file and calls then then() method of the promise as follows:

import promise, { users } from './user.mjs';

function render(users) {

if (!users) {

throw 'The user list is not available.';

}

let list = users

.map((user) => {

return `<li> ${user.name}(<a href="email:${user.email}">${user.email}</a>)</li>`;

})

.join(' ');

return `<ol>${list}</ol>`;

}

promise.then(() => {

let container = document.querySelector('.container');

try {

container.innerHTML = render(users);

} catch (error) {

container.innerHTML = error;

}

});

Code language: JavaScript (javascript)

How it works.

First, import promise and users from the user.mjs module:

import promise, { users } from './user.mjs';

Code language: JavaScript (javascript)

Second, call the then() method of the promise and wait for the API call to complete to use its results:

promise.then(() => {

let container = document.querySelector('.container');

try {

container.innerHTML = render(users);

} catch (error) {

container.innerHTML = error;

}

});

Code language: JavaScript (javascript)

Now, if you open the index.html, you’ll see a list of users. However, you need to know the right promise to wait for it when you use the module.

ES2022 introduc in this workaroundes the top-level await module to resolve this issue.

Using the top-level await

First, change the user.mjs to the following:

const url = 'https://jsonplaceholder.typicode.com/users';

const response = await fetch(url);

let users = await response.json();

export { users };

Code language: JavaScript (javascript)

In this module, you can use the await keyword without placing a statement inside an async function.

Second, import the users from the user.mjs module and use it:

import { users } from './user.mjs';

function render(users) {

if (!users) {

throw 'The user list is not available.';

}

let list = users

.map((user) => {

return `<li> ${user.name}(<a href="email:${user.email}">${user.email}</a>)</li>`;

})

.join(' ');

return `<ol>${list}</ol>`;

}

let container = document.querySelector('.container');

try {

container.innerHTML = render(users);

} catch (error) {

container.innerHTML = error;

}

Code language: JavaScript (javascript)

In this case, the app.mjs module will wait for the user.mjs module to complete before executing its body.

JavaScript top-level await use cases

When do you use the top-level await? Here are some use cases.

Dynamic dependency pathing

const words = await import(`/i18n/${navigator.language}`);

Code language: JavaScript (javascript)

In this example, the top-level await allows modules to use runtime values to decide the dependencies, which is useful for the following scenarios:

* Internationalization (i18n)
* Development / production environment splits.

Dependency fallback

In this case, you can use the top-level await to load a module from a server (cdn1). And if it fails, you can load it from a backup server (cdn2):

let module;

try {

module = await import('https://cdn1.com/module');

} catch {

module = await import('https://cdn2.com/module');

}

Code language: JavaScript (javascript)

Summary

* A top-level await module acts like an async function.
* When a module imports a top-level await module, it waits for the top-level await module to complete before evaluating its body.