**Import & export**

The **export** declaration is used to export values from a JavaScript module. Exported values can then be imported into other programs with the [import](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import) declaration or [dynamic import](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/import). The value of an imported binding is subject to change in the module that exports it — when a module updates the value of a binding that it exports, the update will be visible in its imported value.

In order to use the export declaration in a source file, the file must be interpreted by the runtime as a [module](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules). In HTML, this is done by adding type="module" to the [<script>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script) tag, or by being imported by another module. Modules are automatically interpreted in [strict mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode).

[**Syntax**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export#syntax)

// Exporting declarations

export let name1, name2/\*, … \*/; // also var

export const name1 = 1, name2 = 2/\*, … \*/; // also var, let

export function functionName() { /\* … \*/ }

export class ClassName { /\* … \*/ }

export function\* generatorFunctionName() { /\* … \*/ }

export const { name1, name2: bar } = o;

export const [ name1, name2 ] = array;

// Export list

export { name1, /\* …, \*/ nameN };

export { variable1 as name1, variable2 as name2, /\* …, \*/ nameN };

export { variable1 as "string name" };

export { name1 as default /\*, … \*/ };

// Default exports

export default expression;

export default function functionName() { /\* … \*/ }

export default class ClassName { /\* … \*/ }

export default function\* generatorFunctionName() { /\* … \*/ }

export default function () { /\* … \*/ }

export default class { /\* … \*/ }

export default function\* () { /\* … \*/ }

// Aggregating modules

export \* from "module-name";

export \* as name1 from "module-name";

export { name1, /\* …, \*/ nameN } from "module-name";

export { import1 as name1, import2 as name2, /\* …, \*/ nameN } from "module-name";

export { default, /\* …, \*/ } from "module-name";

Copy to Clipboard

nameN

Identifier to be exported (so that it can be imported via [import](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import) in another script). If you use an alias with as, the actual exported name can be specified as a string literal, which may not be a valid identifier.

[**Description**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export#description)

Every module can have two different types of export, *named export* and *default export*. You can have multiple named exports per module but only one default export. Each type corresponds to one of the above syntax.

Named exports:

// export features declared elsewhere

export { myFunction2, myVariable2 };

// export individual features (can export var, let,

// const, function, class)

export let myVariable = Math.sqrt(2);

export function myFunction() { /\* ... \*/ };

Copy to Clipboard

After the export keyword, you can use let, const, and var declarations, as well as function or class declarations. You can also use the export { name1, name2 } syntax to export a list of names declared elsewhere. Note that export {} does not export an empty object — it's a no-op declaration that exports nothing (an empty name list).

Export declarations are not subject to [temporal dead zone](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#temporal_dead_zone_tdz) rules. You can declare that the module exports X before the name X itself is declared.

export { x };

const x = 1;

// This works, because `export` is only a declaration, but doesn't

// utilize the value of `x`.

Copy to Clipboard

Default exports:

// export feature declared elsewhere as default

export { myFunction as default };

// This is equivalent to:

export default myFunction;

// export individual features as default

export default function () { /\* … \*/ }

export default class { /\* … \*/ }

Copy to Clipboard

**Note:** Names for export declarations must be distinct from each other. Having exports with duplicate names or using more than one default export will result in a [SyntaxError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError) and prevent the module from being evaluated.

The export default syntax allows any expression.

export default 1 + 1;

Copy to Clipboard

As a special case, functions and classes are exported as *declarations*, not expressions, and these declarations can be anonymous. This means functions will be hoisted.

// Works because `foo` is a function declaration,

// not a function expression

foo();

export default function foo() {

console.log("Hi");

}

// It's still technically a declaration, but it's allowed

// to be anonymous

export default function () {

console.log("Hi");

}

Copy to Clipboard

Named exports are useful when you need to export several values. When importing this module, named exports must be referred to by the exact same name (optionally renaming it with as), but the default export can be imported with any name. For example:

// file test.js

const k = 12;

export default k;

Copy to Clipboard

// some other file

import m from './test'; // note that we have the freedom to use import m instead of import k, because k was default export

console.log(m); // will log 12

Copy to Clipboard

You can also rename named exports to avoid naming conflicts:

export {

myFunction as function1,

myVariable as variable,

};

Copy to Clipboard

You can rename a name to something that's not a valid identifier by using a string literal. For example:

export { myFunction as "my-function" };

Copy to Clipboard

[**Re-exporting / Aggregating**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export#re-exporting_aggregating)

A module can also "relay" values exported from other modules without the hassle of writing two separate import/export statements. This is often useful when creating a single module concentrating various exports from various modules (usually called a "barrel module").

This can be achieved with the "export from" syntax:

export {

default as function1,

function2,

} from 'bar.js';

Copy to Clipboard

Which is comparable to a combination of import and export, except that function1 and function2 do not become available inside the current module:

import { default as function1, function2 } from 'bar.js';

export { function1, function2 };

Copy to Clipboard

Most of the "import from" syntaxes have "export from" counterparts.

export { x } from "mod";

export { x as v } from "mod";

export \* as ns from "mod";

Copy to Clipboard

There is also export \* from "mod", although there's no import \* from "mod". This re-exports all **named** exports from mod as the named exports of the current module, but the default export of mod is not re-exported. If there are two wildcard exports statements that implicitly re-export the same name, neither one is re-exported.

// -- mod1.js --

export const a = 1;

// -- mod2.js --

export const a = 3;

// -- barrel.js --

export \* from "./mod1.js";

export \* from "./mod2.js";

// -- main.js --

import \* as ns from "./barrel.js";

console.log(ns.a); // undefined

Copy to Clipboard

Attempting to import the duplicate name directly will throw an error.

import { a } from "./barrel.js";

// SyntaxError: The requested module './barrel.js' contains conflicting star exports for name 'a'

Copy to Clipboard

The following is syntactically invalid despite its import equivalent:

export DefaultExport from 'bar.js'; // Invalid

The correct way of doing this is to rename the export:

export { default as DefaultExport } from 'bar.js';

Copy to Clipboard

The "export from" syntax allows the as token to be omitted, which makes the default export still re-exported as default export.

export { default, function2 } from 'bar.js';

Copy to Clipboard

[**Examples**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export#examples)

[**Using named exports**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export#using_named_exports)

In a module my-module.js, we could include the following code:

// module "my-module.js"

function cube(x) {

return x \* x \* x;

}

const foo = Math.PI + Math.SQRT2;

const graph = {

options: {

color: 'white',

thickness: '2px',

},

draw() {

console.log('From graph draw function');

}

};

export { cube, foo, graph };

Copy to Clipboard

Then in the top-level module included in your HTML page, we could have:

import { cube, foo, graph } from './my-module.js';

graph.options = {

color: 'blue',

thickness: '3px',

};

graph.draw();

console.log(cube(3)); // 27

console.log(foo); // 4.555806215962888

Copy to Clipboard

It is important to note the following:

* You need to include this script in your HTML with a [<script>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script) element of type="module", so that it gets recognized as a module and dealt with appropriately.
* You can't run JS modules via a file:// URL — you'll get [CORS](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS) errors. You need to run it via an HTTP server.

[**Using the default export**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export#using_the_default_export)

If we want to export a single value or to have a fallback value for your module, you could use a default export:

// module "my-module.js"

export default function cube(x) {

return x \* x \* x;

}

Copy to Clipboard

Then, in another script, it is straightforward to import the default export:

import cube from './my-module.js';

console.log(cube(3)); // 27

Copy to Clipboard

[**Using export from**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export#using_export_from)

Let's take an example where we have the following hierarchy:

* childModule1.js: exporting myFunction and myVariable
* childModule2.js: exporting MyClass
* parentModule.js: acting as an aggregator (and doing nothing else)
* top level module: consuming the exports of parentModule.js

This is what it would look like using code snippets:

// In childModule1.js

function myFunction() {

console.log("Hello!");

}

const myVariable = 1;

export { myFunction, myVariable };

Copy to Clipboard

// In childModule2.js

class MyClass {

constructor(x) {

this.x = x;

}

}

export { MyClass };

Copy to Clipboard

// In parentModule.js

// Only aggregating the exports from childModule1 and childModule2

// to re-export them

export { myFunction, myVariable } from 'childModule1.js';

export { MyClass } from 'childModule2.js';

Copy to Clipboard

// In top-level module

// We can consume the exports from a single module since parentModule

// "collected"/"bundled" them in a single source

import { myFunction, myVariable, MyClass } from 'parentModule.js'

Copy to Clipboard