JavaScript modules

This guide gives you all you need to get started with JavaScript module syntax.

A background on modules

JavaScript programs started off pretty small — most of its usage in the early days was to do isolated scripting tasks, providing a bit of interactivity to your web pages where needed, so large scripts were generally not needed. Fast forward a few years and we now have complete applications being run in browsers with a lot of JavaScript, as well as JavaScript being used in other contexts (Node.js, for example).

It has therefore made sense in recent years to start thinking about providing mechanisms for splitting JavaScript programs up into separate modules that can be imported when needed. Node.js has had this ability for a long time, and there are a number of JavaScript libraries and frameworks that enable module usage (for example, other CommonJS and AMD-based module systems like RequireJS, and more recently Webpack and Babel).

The good news is that modern browsers have started to support module functionality natively. This can only be a good thing — browsers can optimize loading of modules, making it more efficient than having to use a library and do all of that extra client-side processing and extra round trips.

Basic example structure

In our first example (see basic-modules) we have a file structure as follows:

```
index.html
main.js
modules/
    canvas.js
    square.js
```

The modules directory's two modules are described below:

- `canvas.js` — contains functions related to setting up the canvas:

    - `create()` — creates a canvas with a specified `width` and `height` inside a wrapper `<div>` with a specified ID, which is itself appended inside a specified parent element. Returns an object containing the canvas's 2D context and the wrapper's ID.
    - `createReportList()` — creates an unordered list appended inside a specified wrapper element, which can be used to output report data into. Returns the list's ID.
- `square.js` — contains:
    - `name` — a constant containing the string 'square'.
    - `draw()` — draws a square on a specified canvas, with a specified size, position, and color. Returns an object containing the square's size, position, and color.
    - `reportArea()` — writes a square's area to a specific report list, given its length.
    - `reportPerimeter()` — writes a square's perimeter to a specific report list, given its length.

Aside — .mjs versus .js

Throughout this article, we've used `.js` extensions for our module files, but in other resources you may see the `.mjs` extension used instead. V8's documentation recommends this, for example. The reasons given are:

- It is good for clarity, i.e. it makes it clear which files are modules, and which are regular JavaScript.
- It ensures that your module files are parsed as a module by runtimes such as Node.js, and build tools such as Babel.

To get modules to work correctly in a browser, you need to make sure that your server is serving them with a `Content-Type` header that contains a JavaScript MIME type such as text/javascript. If you don't, you'll get a strict MIME type checking error along the lines of "The server responded with a non-JavaScript MIME type" and the browser won't run your JavaScript. Most servers already set the correct type for `.js` files, but not yet for `.mjs` files. Servers that already serve `.mjs` files correctly include GitHub Pages and http-server for Node.js.

For learning and portability purposes, we decided to keep to .js.

If you really value the clarity of using .mjs for modules versus using .js for "normal" JavaScript files, but don't want to run into the problem described above, you could always use .mjs during development and convert them to .js during your build step.

It is also worth noting that:

- Some tools may never support `.mjs`.
- The `<script type="module">` attribute is used to denote when a module is being pointed to, as you'll see below.

Exporting module features

The first thing you do to get access to module features is export them. This is done using the export statement.

The easiest way to use it is to place it in front of any items you want exported out of the module, for example:

```
export const name = "square";

export function draw(ctx, length, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, length, length);

  return { length, x, y, color };
}
```

You can export functions, `var`, `let`, `const`, and — as we'll see later — classes. They need to be top-level items; you can't use export inside a function, for example.

A more convenient way of exporting all the items you want to export is to use a single export statement at the end of your module file, followed by a comma-separated list of the features you want to export wrapped in curly braces. For example:

```
export { name, draw, reportArea, reportPerimeter };
```

Importing features into your script

Once you've exported some features out of your module, you need to import them into your script to be able to use them. The simplest way to do this is as follows:

```
import { name, draw, reportArea, reportPerimeter } from "./modules/square.js";
```

You use the `import` statement, followed by a comma-separated list of the features you want to import wrapped in curly braces, followed by the keyword `from`, followed by the *module specifier*.

The module specifier provides a string that the JavaScript environment can resolve to a path to the module file. In a browser, this could be a path relative to the site root, which for our basic-modules example would be `/js-examples/module-examples/basic-modules`. However, here we are instead using the dot `(.)` syntax to mean "the current location", followed by the relative path to the file we are trying to find. This is much better than writing out the entire absolute path each time, as relative paths are shorter and make the URL portable — the example will still work if you move it to a different location in the site hierarchy.

So for example:

```
/js-examples/module-examples/basic-modules/modules/square.js
```

becomes

```
./modules/square.js
```

Once you've imported the features into your script, you can use them just like they were defined inside the same file. The following is found in main.js, below the import lines:

```
const myCanvas = create("myCanvas", document.body, 480, 320);
const reportList = createReportList(myCanvas.id);

const square1 = draw(myCanvas.ctx, 50, 50, 100, "blue");
reportArea(square1.length, reportList);
reportPerimeter(square1.length, reportList);
```

Note: **The imported values are read-only views of the features that were exported**. Similar to **const variables**, you cannot re-assign the variable that was imported, but you can still modify properties of object values. The value can only be re-assigned by the module exporting it. See the import reference for an example.

Importing modules using import maps

Import maps allow developers to instead specify almost any text they want in the module specifier when importing a module; the map provides a corresponding value that will replace the text when the module URL is resolved.

For example, the `imports` key in the import map below defines a "module specifier map" JSON object where the property names can be used as module specifiers, and the corresponding values will be substituted when the browser resolves the module URL. The values must be absolute or relative URLs. Relative URLs are resolved to absolute URL addresses using the base URL of the document containing the import map.

```
<script type="importmap">
  {
    "imports": {
      "shapes": "./shapes/square.js",
      "shapes/square": "./modules/shapes/square.js",
      "https://example.com/shapes/": "/shapes/square/",
      "https://example.com/shapes/square.js": "./shapes/square.js",
      "../shapes/square": "./shapes/square.js"
    }
  }
</script>
```

The import map is defined using a JSON object inside a `<script>` element with the type attribute set to `importmap`. There can only be one import map in the document, and because it is used to resolve which modules are loaded in both static and dynamic imports, it must be declared before any `<script>` elements that import modules.

With this map you can now use the property names above as module specifiers. If there is no trailing forward slash on the module specifier key then the whole module specifier key is matched and substituted. For example, below we match bare module names, and remap a URL to another path.

```
// Bare module names as module specifiers
import { name as squareNameOne } from "shapes";
import { name as squareNameTwo } from "shapes/square";

// Remap a URL to another URL
import { name as squareNameThree } from "https://example.com/shapes/mod/square.js";
```

If the module specifier has a trailing forward slash then the value must have one as well, and the key is matched as a "path prefix". This allows remapping of whole classes of URLs.

```
// Remap a URL as a prefix ( https://example.com/shapes/)
import { name as squareNameFour } from "https://example.com/shapes/square.js";
```

It is possible for multiple keys in an import map to be valid matches for a module specifier. For example, a module specifier of `shapes/circle/` could match the module specifier keys `shapes/` and `shapes/circle/`. In this case the browser will select the most specific (longest) matching module specifier key.

Import maps allow modules to be imported using bare module names (as in Node.js), and can also simulate importing modules from packages, both with and without file extensions. While not shown above, they also allow particular versions of a library to be imported, based on the path of the script that is importing the module.

Feature detection

You can check support for import maps using the `HTMLScriptElement.supports()` static method (which is itself broadly supported):

```
if (HTMLScriptElement.supports?.("importmap")) {
  console.log("Browser supports import maps.");
}
```

Importing modules as bare names

In some JavaScript environments, such as Node.js, you can use bare names for the module specifier. This works because the environment can resolve module names to a standard location in the file system. For example, you might use the following syntax to import the "square" module.

```
import { name, draw, reportArea, reportPerimeter } from "square";
```

To use bare names on a browser you need an import map, which provides the information needed by the browser to resolve module specifiers to URLs (JavaScript will throw a `TypeError` if it attempts to import a module specifier that can't be resolved to a module location).

Below you can see a map that defines a `square` module specifier key, which in this case maps to a relative address value.

```
<script type="importmap">
  {
    "imports": {
      "square": "./shapes/square.js"
    }
  }
</script>
```

With this map we can now use a bare name when we import the module:

```
import { name as squareName, draw } from "square";
```

Remapping module paths

Module specifier map entries, where both the specifier key and its associated value have a trailing forward slash (`/`), can be used as a path-prefix. This allows the remapping of a whole set of import URLs from one location to another.

Packages of modules

The following JSON import map definition maps `lodash` as a bare name, and the module specifier prefix `lodash/` to the path `/node_modules/lodash-es/` (resolved to the document base URL):

```
{
  "imports": {
    "lodash": "/node_modules/lodash-es/lodash.js",
    "lodash/": "/node_modules/lodash-es/"
  }
}
```

With this mapping you can import both the whole "package", using the bare name, and modules within it (using the path mapping):

```
import _ from "lodash";
import fp from "lodash/fp.js";
```

It is possible to import `fp` above without the `.js` file extension, but you would need to create a bare module specifier key for that file, such as `lodash/fp`, rather than using the path. This may be reasonable for just one module, but scales poorly if you wish to import many modules.

General URL remapping

A module specifier key doesn't have to be path — it can also be an absolute URL (or a URL-like relative path like `./`, `../`, `/`). This may be useful if you want to remap a module that has absolute paths to a resource with your own local resources.

```
{
  "imports": {
    "https://www.unpkg.com/moment/": "/node_modules/moment/"
  }
}
```

Scoped modules for version management

Ecosystems like Node use package managers such as npm to manage modules and their dependencies. The package manager ensures that each module is separated from other modules and their dependencies. As a result, while a complex application might include the same module multiple times with several different versions in different parts of the module graph, users do not need to think about this complexity.

Import maps similarly allow you to have multiple versions of dependencies in your application and refer to them using the same module specifier. You implement this with the `scopes` key, which allows you to provide module specifier maps that will be used depending on the path of the script performing the import. The example below demonstrates this.

```
{
  "imports": {
    "coolmodule": "/node_modules/coolmodule/index.js"
```

```
  },
  "scopes": {
    "/node_modules/dependency/": {
      "coolmodule": "/node_modules/some/other/location/coolmodule/index.js"
    }
  }
}
```

With this mapping, if a script with an URL that contains `/node_modules/dependency/` imports `coolmodule`, the version in `/node_modules/some/other/location/coolmodule/index.js` will be used. The map in `imports` is used as a fallback if there is no matching scope in the scoped map, or the matching scopes don't contain a matching specifier. For example, if `coolmodule` is imported from a script with a non-matching scope path, then the module specifier map in imports will be used instead, mapping to the version in `/node_modules/coolmodule/index.js`.

Note that the path used to select a scope does not affect how the address is resolved. The value in the mapped path does not have to match the scopes path, and relative paths are still resolved to the base URL of the script that contains the import map.

Just as for module specifier maps, you can have many scope keys, and these may contain overlapping paths. If multiple scopes match the referrer URL, then the most specific scope path is checked first (the longest scope key) for a matching specifier. The browsers will fall back to the next most specific matching scoped path if there is no matching specifier, and so on. If there is no matching specifier in any of the matching scopes, the browser checks for a match in the module specifier map in the `imports` key.

Scoped modules for version management

Script files used by websites often have hashed filenames to simplify caching. The downside of this approach is that if a module changes, any modules that import it using its hashed filename will also need to be updated/regenerated. This potentially results in a cascade of updates, which is wasteful of network resources.

Import maps provide a convenient solution to this problem. Rather than depending on specific hashed filenames, applications and scripts instead depend on an un-hashed version of the module name (address). An import map like the one below then provides a mapping to the actual script file.

```
{
  "imports": {
    "main_script": "/node/srcs/application-fg7744e1b.js",
```

```
    "dependency_script": "/node/srcs/dependency-3qn7e4b1q.js"
  }
}
```

If `dependency_script` changes, then its hash contained in the file name changes as well. In this case, we only need to update the import map to reflect the changed name of the module. We don't have to update the source of any JavaScript code that depends on it, because the specifier in the import statement does not change.

Applying the module to your HTML

Now we just need to apply the `main.js` module to our HTML page. This is very similar to how we apply a regular script to a page, with a few notable differences.

First of all, you need to include `type="module"` in the `<script>` element, to declare this script as a module. To import the `main.js` script, we use this:

```
<script type="module" src="main.js"></script>
```

You can also embed the module's script directly into the HTML file by placing the JavaScript code within the body of the `<script>` element:

```
<script type="module">
  /* JavaScript module code here */
</script>
```

The script into which you import the module features basically acts as the top-level module. If you omit it, Firefox for example gives you an error of "SyntaxError: import declarations may only appear at top level of a module".

You can only use `import` and `export` statements inside modules, not regular scripts.

Other differences between modules and standard scripts

- You need to pay attention to local testing — if you try to load the HTML file locally (i.e. with a `file://` URL), you'll run into CORS errors due to JavaScript module security requirements. You need to do your testing through a server.

- Also, note that you might get different behavior from sections of script defined inside modules as opposed to in standard scripts. This is because modules use strict mode automatically.

- There is no need to use the `defer` attribute (see <script> attributes) when loading a module script; modules are deferred automatically.

- Modules are only executed once, even if they have been referenced in multiple `<script>` tags.

- Last but not least, let's make this clear — **module features are imported into the scope of a single script — they aren't available in the global scope**. Therefore, you will only be able to access imported features in the script they are imported into, and you won't be able to access them from the JavaScript console, for example. You'll still get syntax errors shown in the DevTools, but you'll not be able to use some of the debugging techniques you might have expected to use.

Module-defined variables are scoped to the module unless explicitly attached to the global object. On the other hand, globally-defined variables are available within the module. For example, given the following code:

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <meta charset="UTF-8" />
    <title></title>
    <link rel="stylesheet" href="" />
  </head>
  <body>
    <div id="main"></div>
    <script>
      // A var statement creates a global variable.
      var text = "Hello";
    </script>
    <script type="module" src="./render.js"></script>
  </body>
</html>
```

```
/* render.js */
document.getElementById("main").innerText = text;
```

The page would still render `Hello`, because the global variables `text` and `document` are available in the module. (Also note from this example that a module doesn't necessarily need an import/export statement — the only thing needed is for the entry point to have `type="module"`.)

Default exports versus named exports

The functionality we've exported so far has been comprised of named exports — each item (be it a function, const, etc.) has been referred to by its name upon export, and that name has been used to refer to it on import as well.

There is also a type of export called the default export — this is designed to make it easy to have a default function provided by a module, and also helps JavaScript modules to interoperate with existing CommonJS and AMD module systems.

Let's look at an example as we explain how it works. In our basic-modules `square.js` you can find a function called `randomSquare()` that creates a square with a random color, size, and position. We want to export this as our default, so at the bottom of the file we write this:

```
export default randomSquare;
```

Note the lack of curly braces.

We could instead prepend export default onto the function and define it as an anonymous function, like this:

```
export default function (ctx) {
  // …
}
```

Over in our `main.js` file, we import the default function using this line:

```
import randomSquare from "./modules/square.js";
```

Again, note the lack of curly braces. This is because there is only one default export allowed per module, and we know that `randomSquare` is it. The above line is basically shorthand for:

```
import { default as randomSquare } from "./modules/square.js";
```

Renaming imports and exports

In JavaScript, you can rename imports and exports using the `as` keyword. This can be useful when you want to use a different name for an imported or exported entity than the one defined in the original module.

Here's an example of how to rename imports and exports using the `as` keyword:

```
// my-module.js
export const foo = 'foo';
export const bar = 'bar';
export const baz = 'baz';
```

In this example, we have a module that exports three variables: `foo`, `bar`, and `baz`. We can import these variables into another file and rename them like this:

```
// main.js
import { foo as myFoo, bar as myBar, baz as myBaz } from './my-module.js';

console.log(myFoo); // logs 'foo'
console.log(myBar); // logs 'bar'
console.log(myBaz); // logs 'baz'
```

In this example, we're importing the variables `foo`, `bar`, and `baz` from `my-module.js` using the `import` statement. We're also renaming each variable using the `as` keyword: `foo` is renamed to `myFoo`, `bar` is renamed to `myBar`, and `baz` is renamed to `myBaz`.

We can also use the `as` keyword to rename entities when exporting from a module:

```
// my-module.js
const foo = 'foo';
const bar = 'bar';
const baz = 'baz';

export { foo as myFoo, bar as myBar, baz as myBaz };
```

In this example, we're exporting the variables `foo`, `bar`, and `baz`, but we're renaming each variable using the `as` keyword: `foo` is renamed to `myFoo`, `bar` is renamed to `myBar`, and `baz` is renamed to `myBaz`. This allows other modules to import these variables with the new names:

```
// main.js
import { myFoo, myBar, myBaz } from './my-module.js';

console.log(myFoo); // logs 'foo'
console.log(myBar); // logs 'bar'
console.log(myBaz); // logs 'baz'
```

In general, renaming imports and exports can be useful for avoiding naming collisions, making code more readable, or adhering to naming conventions.

Creating a module object

In JavaScript, a module is a self-contained unit of code that can be imported and exported to other modules. A module can encapsulate variables, functions, classes, or other entities, making them private and only accessible to other modules through a defined interface.

To create a module object in JavaScript, you can use the `Module` function which is built into modern web browsers. Here's an example of how to create a simple module object:

```
// my-module.js
const privateVariable = 'This variable is private';

function privateFunction() {
  console.log('This function is private');
}

export const publicVariable = 'This variable is public';

export function publicFunction() {
  console.log('This function is public');
}
```

In this example, we've defined a module that contains a private variable and function, as well as a public variable and function. The public entities are exported using the `export` keyword.

To use this module in another file, we can import the public entities using the `import` keyword:

```
// main.js
import { publicVariable, publicFunction } from './my-module.js';

console.log(publicVariable); // logs 'This variable is public'

publicFunction(); // logs 'This function is public'
```

Here, we've imported the `publicVariable` and `publicFunction` entities from `my-module.js` using the `import` statement. We can then use these entities in our main file.

It's worth noting that the `Module` function is not supported in all browsers. If you need to support older browsers, you may need to use a module bundler like Webpack, Rollup, or Parcel to create modules instead. These bundlers can convert your module code into a format that is compatible with older browsers.

Modules and classes

To export a class as a module, you can use the `export` keyword followed by the `class` keyword:

```
// my-class.js
export class MyClass {
  constructor() {
    // constructor code
  }

  myMethod() {
    // method code
  }
}
```

In this example, we're exporting a class called `MyClass` from a module called `my-class.js`. The class has a constructor and a method called `myMethod`.

To import this class in another module, you can use the `import` keyword followed by the class name and the path to the module file:

```
// main.js
import { MyClass } from './my-class.js';
```

```
const myObject = new MyClass();
myObject.myMethod();
```

In this example, we're importing the `MyClass` class from the `my-class.js` module file using the `import` statement. We're then creating a new instance of the class and calling the `myMethod` method.

You can also use the `as` keyword to rename the imported class:

```
// main.js
import { MyClass as MyRenamedClass } from './my-class.js';

const myObject = new MyRenamedClass();
myObject.myMethod();
```

In this example, we're importing the `MyClass` class from the `my-class.js` module file and renaming it to `MyRenamedClass` using the `as` keyword. We're then creating a new instance of the renamed class and calling the `myMethod` method.

Overall, exporting and importing classes as modules in JavaScript follows the same syntax as exporting and importing other types of entities such as functions and variables.

Aggregating modules

Aggregating modules in JavaScript means combining multiple modules into a single, larger module. This can be done to simplify the codebase, reduce the number of requests made to the server, or improve performance.

One common way to aggregate modules is by using an ES6 module bundler like Webpack, Rollup, or Parcel. These bundlers can analyze your code and create a single, optimized JavaScript file that includes all of your modules.

Here's an example:

Suppose we have three modules called `foo.js`, `bar.js`, and `baz.js`:

```
// foo.js
export function foo() {
  console.log('foo');
}

// bar.js
export function bar() {
  console.log('bar');
}

// baz.js
export function baz() {
  console.log('baz');
}
```

We can aggregate these modules into a single module by creating a new module called `all.js` that imports and re-exports the modules:

```
// all.js
export * from './foo.js';
export * from './bar.js';
export * from './baz.js';
```

In this example, we're using the `export *` syntax to re-export all of the functions from each of the three modules. We can then import these functions from `all.js` in our main application file:

```
// main.js
import { foo, bar, baz } from './all.js';

foo(); // logs 'foo'
bar(); // logs 'bar'
baz(); // logs 'baz'
```

By aggregating the three modules into a single file, we've reduced the number of requests made to the server and simplified the codebase.

However, it's worth noting that aggregating modules can lead to larger file sizes, which can negatively impact performance. As with any optimization technique, it's important to weigh the benefits against the costs and to test the performance of your application to ensure that it meets your performance goals.

Dynamic module loading

Dynamic module loading is a feature in JavaScript that allows you to load modules on demand, rather than loading them all at once when the page is initially loaded. This can be particularly useful for large web applications where you may have a large number of modules that aren't always needed, or for cases where you want to reduce initial load times.

The most common way to dynamically load a module in JavaScript is by using the `import()` function, which returns a promise that resolves to the module namespace object. Here's an example:

```
import('./myModule.js')
  .then(module => {
    // Do something with the module
  })
  .catch(error => {
    // Handle any errors that occurred while loading the module
  });
```

In this example, we're using `import()` to load the `myModule.js` file. When the module is loaded, the promise resolves with a reference to the module namespace object, which we can then use to access the module's exports.

It's worth noting that dynamic module loading is a relatively new feature in JavaScript and may not be supported in all browsers. You can check if a browser supports dynamic module loading using the `supportsDynamicImport` property on the `import` object:

```
if ('import' in window && 'supportsDynamicImport' in window.import) {
  // Dynamic module loading is supported
} else {
  // Dynamic module loading is not supported
}
```

Overall, dynamic module loading is a powerful feature in JavaScript that can help you improve the performance of your web applications by only loading the modules that you need, when you need them.

Top level await

Top-level `await` is a feature in ECMAScript 2021 (ES2021) that allows you to use `await` at the top level of a module, outside of an `async` function. Prior to ES2021, `await` could only be used inside an `async` function.

With top-level `await`, you can use `await` to wait for a promise to resolve before continuing execution of the module. This can simplify your code and make it easier to work with asynchronous operations.

Here's an example of how to use top-level `await` in a JavaScript module:

```
// my-module.js
const myPromise = new Promise(resolve => {
  setTimeout(() => {
    resolve('Hello, world!');
  }, 1000);
});

export default async function() {
  const result = await myPromise;
  console.log(result);
}
```

In this example, we're defining a module that exports an `async` function. Inside the function, we're defining a promise that resolves with the string 'Hello, world!' after a 1 second delay. We're then using `await` to wait for the promise to resolve and logging the result to the console.

We can then import and use the module in another file like this:

```
// main.js
import myFunction from './my-module.js';

await myFunction();
```

In this example, we're using top-level `await` to wait for the `myFunction` function to complete before continuing execution of the `main.js` module.

It's worth noting that top-level `await` is only available in modules, not in scripts. If you want to use top-level `await`, you need to use an ES module with an `import` statement.

Cyclic imports

Cyclic imports, also known as circular dependencies, occur when two or more modules depend on each other in a way that creates an endless loop. This can cause issues in your JavaScript code and can make it difficult to understand and debug.

To understand cyclic imports, let's start by looking at how modules work in JavaScript. In the CommonJS module system (which is used in Node.js), a module can export values using the `module.exports` object:

```
// moduleA.js
const moduleB = require('./moduleB');

module.exports = {
  a: 1,
  b: moduleB
};

// moduleB.js
const moduleA = require('./moduleA');

module.exports = {
  c: 2,
  d: moduleA
};
```

In this example, `moduleA` requires `moduleB`, and `moduleB` requires `moduleA`. This creates a cyclic dependency between the two modules. When you try to run this code, you'll get an error like this:

```
Error: Cannot find module './moduleA'
```

This error occurs because Node.js tries to load `moduleA` when it is required by `moduleB`, but it has not finished loading `moduleA` yet. This creates an endless loop and causes a stack overflow.

To fix this issue, you need to break the cycle. One way to do this is to restructure your code so that the modules don't depend on each other directly. You can move the common functionality to a separate module and have both modules depend on it:

```
// common.js
module.exports = {
  sharedFunction: function() {}
};

// moduleA.js
const common = require('./common');
const moduleB = require('./moduleB');

module.exports = {
  a: 1,
  b: moduleB,
  sharedFunction: common.sharedFunction
};

// moduleB.js
const common = require('./common');
const moduleA = require('./moduleA');

module.exports = {
  c: 2,
  d: moduleA,
  sharedFunction: common.sharedFunction
};
```

In this example, we've created a separate module common that exports a shared function. Both `moduleA` and `moduleB` require common and use its `sharedFunction`. This way, there is no longer a direct circular dependency between `moduleA` and `moduleB`.

Another way to break the cycle is to use dynamic imports (two lessons back – "Dynamic module loading"). Dynamic imports are using `.then` keyword which allows you to import a module only when you need it, rather than at the top of your code.

You should usually avoid cyclic imports in your project, because they make your code more error-prone. Some common cycle-elimination techniques are:

- Merge the two modules into one.
- Move the shared code into a third module.
- Move some code from one module to the other.

However, cyclic imports can also occur if the libraries depend on each other, which is harder to fix.

Authoring "isomorphic" modules

An isomorphic module is a JavaScript module that can run in both client-side (browser) and server-side (Node.js) environments. This is also known as writing "universal" or "cross-platform" JavaScript code.

To author an isomorphic module, you need to take into account the differences between the browser and Node.js environments. For example, in the browser, you can't use modules that depend on Node.js-specific APIs, like the `fs` (file system) module. Similarly, in Node.js, you can't use modules that depend on browser-specific APIs, like the `window` object.

To overcome these differences, you can use techniques like conditional imports and exports, feature detection, and environment-specific configuration. Here's an example of a simple isomorphic module:

```
// isomorphicModule.js

let isBrowser = false;

if (typeof window !== "undefined") {
  isBrowser = true;
}

let fs;

if (!isBrowser) {
  fs = require("fs");
}

export function readFromFile(path) {
  if (isBrowser) {
    // use browser-specific code to read file
  } else {
    // use Node.js-specific code to read file
    return fs.readFileSync(path, "utf-8");
  }
}

export function writeToConsole(data) {
  if (isBrowser) {
    console.log(data);
  } else {
    console.error(data);
  }
}
```

In this example, we're checking whether the module is running in a browser or in Node.js by looking for the existence of the `window` object. We're also conditionally importing the `fs` module if we're running in Node.js, since it's not available in the browser.

We're exporting two functions: `readFromFile` and `writeToConsole`. The `readFromFile` function uses environment-specific code to read a file, while `writeToConsole` writes data to the console using console.log in the browser and `console.error` in Node.js.

By using these techniques, you can create modules that work seamlessly in both browser and Node.js environments.

Troubleshooting

Here are a few tips that may help you if you are having trouble getting your modules to work. Feel free to add to the list if you discover more!

- We mentioned this before, but to reiterate: `.mjs` files need to be loaded with a MIME-type of `text/javascript` (or another JavaScript-compatible MIME-type, but `text/javascript` is recommended), otherwise you'll get a strict MIME type checking error like "The server responded with a non-JavaScript MIME type".

- If you try to load the HTML file locally (i.e. with a `file://` URL), you'll run into CORS errors due to JavaScript module security requirements. You need to do your testing through a server. GitHub pages is ideal as it also serves `.mjs` files with the correct MIME type.

- Because `.mjs` is a non-standard file extension, some operating systems might not recognize it, or try to replace it with something else. For example, we found that macOS was silently adding on `.js` to the end of `.mjs` files and then automatically hiding the file extension. So all of our files were actually coming out as `x.mjs.js`. Once we turned off automatically hiding file extensions, and trained it to accept `.mjs`, it was OK.