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, and this is what this article is all about. 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.

Browser support

Use of native JavaScript modules is dependent on the `import` and `export` statements; these are supported in browsers as follows:

import

Update compatibility data on GitHub

`import`	
Chrome	61
Edge	16
Firefox	60
IE	No
Opera	48

Safari	10.1
WebView Android	61
Chrome Android	61
Firefox Android	60
Opera Android	45
Safari iOS	10.3
Samsung Internet Android	8.0
nodejs	13.2.0

Dynamic import

Chrome	63
Edge	79
Firefox	67
IE	No
Opera	50
Safari	11.1
WebView Android	63
Chrome Android	63
Firefox Android	67
Opera Android	46
Safari iOS	11.3
Samsung Internet Android	8.0
nodejs	13.2.0

Available in workers

Chrome	80
Edge	80
Firefox	No
IE	No
Opera	No
Safari	No
WebView Android	80
Chrome Android	80

Firefox Android	No
Opera Android	No
Safari iOS	No
Samsung Internet Android	No
nodejs	No

What are we missing?

.. Full support

.. No support

See implementation notes.
User must explicitly enable this feature.

export

Update compatibility data on GitHub

`export`

Chrome	61
Edge	~~16~~
Firefox	~~60~~
IE	No
Opera	48
Safari	10.1
WebView Android	No
Chrome Android	61
Firefox Android	~~60~~
Opera Android	45
Safari iOS	10.3
Samsung Internet Android	8.0
nodejs	~~13.2.0~~

`default` keyword with `export`

Chrome	61
Edge	16
Firefox	60
IE	No
Opera	48
Safari	10.1
WebView Android	No
Chrome Android	61
Firefox Android	60
Opera Android	45
Safari iOS	10.3
Samsung Internet Android	8.0
nodejs	13.2.0

`export * as namespace`

Chrome	72
Edge	79
Firefox	80
IE	No
Opera	60
Safari	No
WebView Android	No
Chrome Android	72
Firefox Android	No
Opera Android	51
Safari iOS	No
Samsung Internet Android	11.0
nodejs	12.0.0

.. Full support

.. No support

See implementation notes.

User must explicitly enable this feature.

Introducing an example

To demonstrate usage of modules, we've created a simple set of examples that you can find on GitHub. These examples demonstrate a simple set of modules that create a `<canvas>` element on a webpage, and then draw (and report information about) different shapes on the canvas.

These are fairly trivial, but have been kept deliberately simple to demonstrate modules clearly.

> **Note**: If you want to download the examples and run them locally, you'll need to run them through a local web server.

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
```

> **Note**: All of the examples in this guide have basically the same structure; the above should start getting pretty familiar.

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.

However, we decided to keep to using `.js`, at least for the moment. 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.

This is OK if you are using such an environment already, or if you aren't but you know what you are doing and have access (i.e. you can configure your server to set the correct `Content-Type` for `.mjs` files). It could however cause confusion if you don't control the server you are serving files from, or are publishing files for public use, as we are here.

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`, such as TypeScript.

- 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: length,
    x: x,
    y: y,
    color: 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/squa
```

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 path to the module file — a path relative to the site root, which for our `basic-modules` example would be `/js-examples/modules/basic-modules`.

However, we've written the path a bit differently — we are using the dot (`.`) syntax to mean "the current location", followed by the path beyond that to the file we are trying to find. This is much better than writing out the entire relative path each time, as it is shorter, and it makes 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/modules/basic-modules/modules/square.js
```

becomes

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

You can see such lines in action in `main.js`.

> **Note**: In some module systems, you can omit the file extension and the dot (e.g. `'/modules/square'`). This doesn't work in native JavaScript modules.

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:

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

let square1 = draw(myCanvas.ctx, 50, 50, 100, 'blue');
```

```
reportArea(square1.length, reportList);
reportPerimeter(square1.length, reportList);
```

Note: Although imported features are available in the file, they are read only views of the feature that was exported. You cannot change the variable that was imported, but you can still modify properties similar to `const`. Additionally, these features are imported as live bindings, meaning that they can change in value even if you cannot modify the binding unlike `const`.

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 modeautomatically.
- 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.

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 (as explained nicely in ES6 In Depth: Modules by Jason Orendorff; search for "Default exports").

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';
```

> **Note**: The as syntax for renaming exported items is explained below in the Renaming imports and exports section.

Avoiding naming conflicts

So far, our canvas shape drawing modules seem to be working OK. But what happens if we try to add a module that deals with drawing another shape, like a circle or triangle? These shapes would probably have associated functions like `draw()`, `reportArea()`, etc. too; if we tried to import different functions of the same name into the same top-level module file, we'd end up with conflicts and errors.

Fortunately there are a number of ways to get around this. We'll look at these in the following sections.

Renaming imports and exports

Inside your `import` and `export` statement's curly braces, you can use the keyword `as` along with a new feature name, to change the identifying name you will use for a feature inside the top-level module.

So for example, both of the following would do the same job, albeit in a slightly different way:

```
// inside module.js
export {
  function1 as newFunctionName,
  function2 as anotherNewFunctionName
};

// inside main.js
import { newFunctionName, anotherNewFunctionName } from './modules/modu
```

```
// inside module.js
export { function1, function2 };

// inside main.js
import { function1 as newFunctionName,
         function2 as anotherNewFunctionName } from './modules/module.j
```

Let's look at a real example. In our renaming directory you'll see the same module system as in the previous example, except that we've added `circle.js` and `triangle.js` modules to draw and report on circles and triangles.

Inside each of these modules, we've got features with the same names being exported, and therefore each has the same `export` statement at the bottom:

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

When importing these into `main.js`, if we tried to use

```
import { name, draw, reportArea, reportPerimeter } from './modules/squa
import { name, draw, reportArea, reportPerimeter } from './modules/circ
import { name, draw, reportArea, reportPerimeter } from './modules/tria
```

The browser would throw an error such as "SyntaxError: redeclaration of import name" (Firefox).

Instead we need to rename the imports so that they are unique:

```
import { name as squareName,
         draw as drawSquare,
         reportArea as reportSquareArea,
         reportPerimeter as reportSquarePerimeter } from './modules/squ

import { name as circleName,
         draw as drawCircle,
         reportArea as reportCircleArea,
         reportPerimeter as reportCirclePerimeter } from './modules/cir

import { name as triangleName,
        draw as drawTriangle,
        reportArea as reportTriangleArea,
        reportPerimeter as reportTrianglePerimeter } from './modules/tr
```

Note that you could solve the problem in the module files instead, e.g.

```
// in square.js
export { name as squareName,
         draw as drawSquare,
         reportArea as reportSquareArea,
         reportPerimeter as reportSquarePerimeter };
```

```
// in main.js
import { squareName, drawSquare, reportSquareArea, reportSquarePerimete
```

And it would work just the same. What style you use is up to you, however it arguably makes more sense to leave your module code alone, and make the changes in the imports. This especially makes sense when you are importing from third party modules that you don't have any control over.

Creating a module object

The above method works OK, but it's a little messy and longwinded. An even better solution is to import each module's features inside a module object. The following syntax form does that:

```
import * as Module from './modules/module.js';
```

This grabs all the exports available inside `module.js`, and makes them available as members of an object `Module`, effectively giving it its own namespace. So for example:

```
Module.function1()
Module.function2()
etc.
```

Again, let's look at a real example. If you go to our module-objects directory, you'll see the same example again, but rewritten to take advantage of this new syntax. In the modules, the exports are all in the following simple form:

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

The imports on the other hand look like this:

```
import * as Canvas from './modules/canvas.js';

import * as Square from './modules/square.js';
import * as Circle from './modules/circle.js';
import * as Triangle from './modules/triangle.js';
```

In each case, you can now access the module's imports underneath the specified object name, for example:

```
let square1 = Square.draw(myCanvas.ctx, 50, 50, 100, 'blue');
Square.reportArea(square1.length, reportList);
Square.reportPerimeter(square1.length, reportList);
```

So you can now write the code just the same as before (as long as you include the object names where needed), and the imports are much neater.

Modules and classes

As we hinted at earlier, you can also export and import classes; this is another option for avoiding conflicts in your code, and is especially useful if you've already got your module code written in an object-oriented style.

You can see an example of our shape drawing module rewritten with ES classes in our classesdirectory. As an example, the `square.js` file now contains all its functionality in a single class:

```
class Square {
  constructor(ctx, listId, length, x, y, color) {
    ...
  }

  draw() {
    ...
  }

  ...
}
```

which we then export:

```
export { Square };
```

Over in `main.js`, we import it like this:

```
import { Square } from './modules/square.js';
```

And then use the class to draw our square:

```
let square1 = new Square(myCanvas.ctx, myCanvas.listId, 50, 50, 100, 'b
square1.draw();
square1.reportArea();
square1.reportPerimeter();
```

Aggregating modules

There will be times where you'll want to aggregate modules together. You might have multiple levels of dependencies, where you want to simplify things, combining several submodules into one parent module. This is possible using export syntax of the following forms in the parent module:

```
export * from 'x.js'
export { name } from 'x.js'
```

For an example, see our module-aggregation directory. In this example (based on our earlier classes example) we've got an extra module called `shapes.js`, which aggregates all the functionality from `circle.js`, `square.js`, and `triangle.js` together. We've also moved our submodules inside a subdirectory inside the `modules` directory called `shapes`. So the module structure in this example is:

```
modules/
  canvas.js
  shapes.js
  shapes/
    circle.js
    square.js
    triangle.js
```

In each of the submodules, the export is of the same form, e.g.

```
export { Square };
```

Next up comes the aggregation part. Inside `shapes.js`, we include the following lines:

```
export { Square } from './shapes/square.js';
export { Triangle } from './shapes/triangle.js';
export { Circle } from './shapes/circle.js';
```

These grab the exports from the individual submodules and effectively make them available from the `shapes.js` module.

> **Note**: The exports referenced in `shapes.js` basically get redirected through the file and don't really exist there, so you won't be able to write any useful related code inside the same file.

So now in the `main.js` file, we can get access to all three module classes by replacing

```
import { Square } from './modules/square.js';
import { Circle } from './modules/circle.js';
import { Triangle } from './modules/triangle.js';
```

with the following single line:

```
import { Square, Circle, Triangle } from './modules/shapes.js';
```

Dynamic module loading

The newest part of the JavaScript modules functionality to be available in browsers is dynamic module loading. This allows you to dynamically load modules only when they are needed, rather than having to load everything up front. This has some obvious performance advantages; let's read on and see how it works.

This new functionality allows you to call `import()` as a function, passing it the path to the module as a parameter. It returns a `Promise`, which fulfills with a module object (see Creating a module object) giving you access to that object's exports, e.g.

```
import('./modules/myModule.js')
  .then((module) => {
    // Do something with the module.
  });
```

Let's look at an example. In the dynamic-module-imports directory we've got another example based on our classes example. This time however we are not drawing anything on the canvas when the example loads. Instead, we include three buttons — "Circle", "Square", and "Triangle" —

that, when pressed, dynamically load the required module and then use it to draw the associated shape.

In this example we've only made changes to our `index.html` and `main.js` files — the module exports remain the same as before.

Over in `main.js` we've grabbed a reference to each button using a `Document.querySelector()` call, for example:

```
let squareBtn = document.querySelector('.square');
```

We then attach an event listener to each button so that when pressed, the relevant module is dynamically loaded and used to draw the shape:

```
squareBtn.addEventListener('click', () => {
  import('./modules/square.js').then((Module) => {
    let square1 = new Module.Square(myCanvas.ctx, myCanvas.listId, 50,
    square1.draw();
    square1.reportArea();
    square1.reportPerimeter();
  })
});
```

Note that, because the promise fulfillment returns a module object, the class is then made a subfeature of the object, hence we now need to access the constructor with `Module.` prepended to it, e.g. `Module.Square( ... ).`

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: `.js` 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 `.js` files with the correct MIME type.

- Because `.mjs` is a non-standard file extension, some operating systems might not recognise 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.