Encapsulating Code With Modules

JavaScript's "shared everything" approach to loading code is one of the most error-prone and confusing aspects of the language. Other languages use concepts such as packages to define code scope, but before ECMAScript 6, everything defined in every JavaScript file of an application shared one global scope. As web applications became more complex and started using even more JavaScript code, that approach caused problems like naming collisions and security concerns. One goal of ECMAScript 6 was to solve the scope problem and bring some order to JavaScript applications. That's where modules come in.

What are Modules?

Modules are JavaScript files that are loaded in a different mode (as opposed to *scripts*, which are loaded in the original way JavaScript worked). This different mode is necessary because modules have very different semantics than scripts:

1. Module code automatically runs in strict mode, and there's no way to opt-out of strict mode.
2. Variables created in the top level of a module aren't automatically added to the shared global scope. They exist only within the top-level scope of the module.
3. The value of `this` in the top level of a module is `undefined`.
4. Modules don't allow HTML-style comments within code (a leftover feature from JavaScript's early browser days).
5. Modules must export anything that should be available to code outside of the module.
6. Modules may import bindings from other modules.

These differences may seem small at first glance, but they represent a significant change in how JavaScript code is loaded and evaluated, which I will discuss over the course of this chapter. The real power of modules is the ability to export and import only bindings you need, rather than everything in a file. A good understanding of exporting and importing is fundamental to understanding how modules differ from scripts.

Basic Exporting

You can use the `export` keyword to expose parts of published code to other modules. In the simplest case, you can place `export` in front of any variable, function, or class declaration to export it from the module, like this:

```
// export data
export var color = "red";
export let name = "Nicholas";
export const magicNumber = 7;

// export function
export function sum(num1, num2) {
    return num1 + num1;
}

// export class
export class Rectangle {
```

```
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }
}

// this function is private to the module
function subtract(num1, num2) {
    return num1 - num2;
}

// define a function...
function multiply(num1, num2) {
    return num1 * num2;
}

// ...and then export it later
export { multiply };
```

There are a few things to notice in this example. First, apart from the `export` keyword, every declaration is exactly the same as it would be otherwise. Each exported function or class also has a name; that's because exported function and class declarations require a name. You can't export anonymous functions or classes using this syntax unless you use the `default` keyword (discussed in detail in the "Default Values in Modules" section).

Next, consider the `multiply()` function, which isn't exported when it's defined. That works because you need not always export a declaration: you can also export references. Finally, notice that this example doesn't export the `subtract()` function. That function won't be accessible from outside this module because any variables, functions, or classes that are not explicitly exported remain private to the module.

Basic Importing

Once you have a module with exports, you can access the functionality in another module by using the `import` keyword. The two parts of an `import` statement are the identifiers you're importing and the module from which those identifiers should be imported. This is the statement's basic form:

```
import { identifier1, identifier2 } from "./example.js";
```

The curly braces after `import` indicate the bindings to import from a given module. The keyword `from` indicates the module from which to import the given binding. The module is specified by a string representing the path to the module (called the *module specifier*). Browsers use the same path format you might pass to the `<script>` element, which means you must include a file extension. Node.js, on the other hand, follows its traditional convention of differentiating between local files and packages based on a filesystem prefix. For example, `example` would be a package and `./example.js` would be a local file.

I> The list of bindings to import looks similar to a destructured object, but it isn't one.

When importing a binding from a module, the binding acts as if it were defined using `const`. That means you can't define another variable with the same name (including importing another binding of the same name), use the identifier before the `import` statement, or change its value.

Importing a Single Binding

Suppose that the first example in the "Basic Exporting" section is in a module with the filename `example.js`. You can import and use bindings from that module in a number of ways. For instance, you can just import one identifier:

```
// import just one
import { sum } from "./example.js";

console.log(sum(1, 2));     // 3

sum = 1;        // error
```

Even though `example.js` exports more than just that one function this example imports only the `sum()` function. If you try to assign a new value to `sum`, the result is an error, as you can't reassign imported bindings.

W> Make sure to include `/`, `./`, or `../` at the beginning of the file you're importing for best compatibility across browsers and Node.js.

Importing Multiple Bindings

If you want to import multiple bindings from the example module, you can explicitly list them out as follows:

```
// import multiple
import { sum, multiply, magicNumber } from "./example.js";
console.log(sum(1, magicNumber));   // 8
console.log(multiply(1, 2));        // 2
```

Here, three bindings are imported from the example module: `sum`, `multiply`, and `magicNumber`. They are then used as if they were locally defined.

Importing All of a Module

There's also a special case that allows you to import the entire module as a single object. All of the exports are then available on that object as properties. For example:

```
// import everything
import * as example from "./example.js";
console.log(example.sum(1,
        example.magicNumber));          // 8
console.log(example.multiply(1, 2));    // 2
```

In this code, all exported bindings in `example.js` are loaded into an object called `example`. The named exports (the `sum()` function, the `multiple()` function, and `magicNumber`) are then accessible as properties on `example`. This import format is called a *namespace import* because the `example` object doesn't exist inside of the `example.js` file and is instead created to be used as a namespace object for all of the exported members of `example.js`.

Keep in mind, however, that no matter how many times you use a module in `import` statements, the module will only be executed once. After the code to import the module executes, the instantiated module is kept in memory and reused whenever another `import` statement references it. Consider the following:

```
import { sum } from "./example.js";
import { multiply } from "./example.js";
import { magicNumber } from "./example.js";
```

Even though there are three `import` statements in this module, `example.js` will only be executed once. If other modules in the same application were to import bindings from `example.js`, those modules would use the same module instance this code uses.

A> ### Module Syntax Limitations A> A> An important limitation of both `export` and `import` is that they must be used outside other statements and functions. For instance, this code will give a syntax error: A> A> `js A> if (flag) { A> export flag; // syntax error A> } A>` A>The `export` statement is inside an `if` statement, which isn't allowed. Exports cannot be conditional or done dynamically in any way. One reason module syntax exists is to let the JavaScript engine staticly determine what will be exported. As such, you can only use `export` at the top-level of a module. A> A> Similarly, you can't use `import` inside of a statement; you can only use it at the top-level. That means this code also gives a syntax error: A> A> `js A> function tryImport() { A> import flag from "./example.js"; // syntax error A> } A>` A> A> You can't dynamically import bindings for the same reason you can't dynamically export bindings. The `export` and `import` keywords are designed to be static so that tools like text editors can easily tell what information is available from a module.

A Subtle Quirk of Imported Bindings

ECMAScript 6's `import` statements create read-only bindings to variables, functions, and classes rather than simply referencing the original bindings like normal variables. Even though the module that imports the binding can't change its value, the module that exports that identifier can. For example, suppose you want to use this module:

```
export var name = "Nicholas";
export function setName(newName) {
    name = newName;
}
```

When you import those two bindings, the `setName()` function can change the value of `name`:

```
import { name, setName } from "./example.js";

console.log(name);        // "Nicholas"
setName("Greg");
console.log(name);        // "Greg"

name = "Nicholas";        // error
```

The call to `setName("Greg")` goes back into the module from which `setName()` was exported and executes there, setting `name` to `"Greg"` instead. Note this change is automatically reflected on the imported `name` binding. That's because `name` is the local name for the exported `name` identifier. The `name` used in the code above and the `name` used in the module being imported from aren't the same.

Renaming Exports and Imports

Sometimes, you may not want to use the original name of a variable, function, or class you've imported from a module. Fortunately, you can change the name of an export both during the export and during the import.

In the first case, suppose you have a function that you'd like to export with a different name. You can use the `as` keyword to specify the name that the function should be known as outside of the module:

```
function sum(num1, num2) {
    return num1 + num2;
}

export { sum as add };
```

Here, the `sum()` function (`sum` is the *local name*) is exported as `add()` (`add` is the *exported name*). That means when another module wants to import this function, it will have to use the name `add` instead:

```
import { add } from "./example.js";
```

If the module importing the function wants to use a different name, it can also use `as`:

```
import { add as sum } from "./example.js";
console.log(typeof add);            // "undefined"
console.log(sum(1, 2));             // 3
```

This code imports the `add()` function using the *import name* and renames it to `sum()` (the local name). That means there is no identifier named `add` in this module.

Default Values in Modules

The module syntax is really optimized for exporting and importing default values from modules, as this pattern was quite common in other module systems, like CommonJS (another JavaScript module format popularized by Node.js). The *default value* for a module is a single variable, function, or class as specified by the `default` keyword, and you can only set one default export per module. Using the `default` keyword with multiple exports is a syntax error.

Exporting Default Values

Here's a simple example that uses the `default` keyword:

```
export default function(num1, num2) {
    return num1 + num2;
}
```

This module exports a function as its default value. The `default` keyword indicates that this is a default export. The function doesn't require a name because the module itself represents the function.

You can also specify an identifier as the default export by placing it after `export default`, such as:

```
function sum(num1, num2) {
    return num1 + num2;
}

export default sum;
```

Here, the `sum()` function is defined first and later exported as the default value of the module. You may want to choose this approach if the default value needs to be calculated.

A third way to specify an identifier as the default export is by using the renaming syntax as follows:

```
function sum(num1, num2) {
    return num1 + num2;
}

export { sum as default };
```

The identifier `default` has special meaning in a renaming export and indicates a value should be the default for the module. Because `default` is a keyword in JavaScript, it can't be used for a variable, function, or class name (it can be used as a property name). So the use of `default` to rename an export is a special case to create a consistency with how non-default exports are defined. This syntax is useful if you want to use a single `export` statement to specify multiple exports, including the default, at once.

Importing Default Values

You can import a default value from a module using the following syntax:

```
// import the default
import sum from "./example.js";

console.log(sum(1, 2));     // 3
```

This import statement imports the default from the module `example.js`. Note that no curly braces are used, unlike you'd see in a non-default import. The local name `sum` is used to represent whatever default function the module exports. This syntax is the cleanest, and the creators of ECMAScript 6 expect it to be the dominant form of import on the Web, allowing you to use an already-existing object.

For modules that export both a default and one or more non-default bindings, you can import all exported bindings with one statement. For instance, suppose you have this module:

```
export let color = "red";

export default function(num1, num2) {
    return num1 + num2;
}
```

You can import both `color` and the default function using the following `import` statement:

```
import sum, { color } from "./example.js";

console.log(sum(1, 2));     // 3
console.log(color);         // "red"
```

The comma separates the default local name from the non-defaults, which are also surrounded by curly braces. Keep in mind that the default must come before the non-defaults in the `import` statement.

As with exporting defaults, you can import defaults with the renaming syntax, too:

```
// equivalent to previous example
import { default as sum, color } from "example";

console.log(sum(1, 2));     // 3
console.log(color);         // "red"
```

In this code, the default export (`default`) is renamed to `sum` and the additional `color` export is also imported. This example is equivalent to the preceding example.

Re-exporting a Binding

There may be a time when you'd like to re-export something that your module has imported (for instance, if you're creating a library out of several small modules). You can re-export an imported value with the patterns

already discussed in this chapter as follows:

```
import { sum } from "./example.js";
export { sum }
```

That works, but a single statement can also do the same thing:

```
export { sum } from "./example.js";
```

This form of `export` looks into the specified module for the declaration of `sum` and then exports it. Of course, you can also choose to export a different name for the same value:

```
export { sum as add } from "./example.js";
```

Here, `sum` is imported from `"./example.js"` and then exported as `add`.

If you'd like to export everything from another module, you can use the `*` pattern:

```
export * from "./example.js";
```

When you export everything, you are including all named exports and excluding any default export. For instance, if `example.js` has a default export, you would need to import it explicitly and then export it explicitly.

Importing Without Bindings

Some modules may not export anything, and instead, only make modifications to objects in the global scope. Even though top-level variables, functions, and classes inside modules don't automatically end up in the global scope, that doesn't mean modules cannot access the global scope. The shared definitions of built-in objects such as `Array` and `Object` are accessible inside a module and changes to those objects will be reflected in other modules.

For instance, if you want to add a `pushAll()` method to all arrays, you might define a module like this:

```
// module code without exports or imports
Array.prototype.pushAll = function(items) {

    // items must be an array
    if (!Array.isArray(items)) {
        throw new TypeError("Argument must be an array.");
    }

    // use built-in push() and spread operator
```

```
    return this.push(...items);
};
```

This is a valid module even though there are no exports or imports. This code can be used both as a module and a script. Since it doesn't export anything, you can use a simplified import to execute the module code without importing any bindings:

```
import "./example.js";

let colors = ["red", "green", "blue"];
let items = [];

items.pushAll(colors);
```

This code imports and executes the module containing the `pushAll()` method, so `pushAll()` is added to the array prototype. That means `pushAll()` is now available for use on all arrays inside of this module.

I> Imports without bindings are most likely to be used to create polyfills and shims.

Loading Modules

While ECMAScript 6 defines the syntax for modules, it doesn't define how to load them. This is part of the complexity of a specification that's supposed to be agnostic to implementation environments. Rather than trying to create a single specification that would work for all JavaScript environments, ECMAScript 6 specifies only the syntax and abstracts out the loading mechanism to an undefined internal operation called `HostResolveImportedModule`. Web browsers and Node.js are left to decide how to implement `HostResolveImportedModule` in a way that makes sense for their respective environments.

Using Modules in Web Browsers

Even before ECMAScript 6, web browsers had multiple ways of including JavaScript in an web application. Those script loading options are:

1. Loading JavaScript code files using the `<script>` element with the `src` attribute specifying a location from which to load the code.
2. Embedding JavaScript code inline using the `<script>` element without the `src` attribute.
3. Loading JavaScript code files to execute as workers (such as a web worker or service worker).

In order to fully support modules, web browsers had to update each of these mechanisms. These details are defined in the HTML specification, and I'll summarize them in this section.

Using Modules With `<script>`

The default behavior of the `<script>` element is to load JavaScript files as scripts (not modules). This happens when the `type` attribute is missing or when the `type` attribute contains a JavaScript content type (such as `"text/javascript"`). The `<script>` element can then execute inline code or load the file specified in `src`. To support modules, the `"module"` value was added as a `type` option. Setting `type` to

`"module"` tells the browser to load any inline code or code contained in the file specified by `src` as a module instead of a script. Here's a simple example:

```
<!-- load a module JavaScript file -->
<script type="module" src="module.js"></script>

<!-- include a module inline -->
<script type="module">

import { sum } from "./example.js";

let result = sum(1, 2);

</script>
```

The first `<script>` element in this example loads an external module file using the `src` attribute. The only difference from loading a script is that `"module"` is given as the `type`. The second `<script>` element contains a module that is embedded directly in the web page. The variable `result` is not exposed globally because it exists only within the module (as defined by the `<script>` element) and is therefore not added to `window` as a property.

As you can see, including modules in web pages is fairly simple and similar to including scripts. However, there are some differences in how modules are loaded.

I> You may have noticed that `"module"` is not a content type like the `"text/javascript"` type. Module JavaScript files are served with the same content type as script JavaScript files, so it's not possible to differentiate solely based on content type. Also, browsers ignore `<script>` elements when the `type` is unrecognized, so browsers that don't support modules will automatically ignore the `<script type="module">` line, providing good backwards-compatibility.

Module Loading Sequence in Web Browsers

Modules are unique in that, unlike scripts, they may use `import` to specify that other files must be loaded to execute correctly. To support that functionality, `<script type="module">` always acts as if the `defer` attribute is applied.

The `defer` attribute is optional for loading script files but is always applied for loading module files. The module file begins downloading as soon as the HTML parser encounters `<script type="module">` with a `src` attribute but doesn't execute until after the document has been completely parsed. Modules are also executed in the order in which they appear in the HTML file. That means the first `<script type="module">` is always guaranteed to execute before the second, even if one module contains inline code instead of specifying `src`. For example:

```
<!-- this will execute first -->
<script type="module" src="module1.js"></script>

<!-- this will execute second -->
<script type="module">
```

```
import { sum } from "./example.js";

let result = sum(1, 2);
</script>

<!-- this will execute third -->
<script type="module" src="module2.js"></script>
```

These three `<script>` elements execute in the order they are specified, so `module1.js` is guaranteed to execute before the inline module, and the inline module is guaranteed to execute before `module2.js`.

Each module may `import` from one or more other modules, which complicates matters. That's why modules are parsed completely first to identify all `import` statements. Each `import` statement then triggers a fetch (either from the network or from the cache), and no module is executed until all `import` resources have first been loaded and executed.

All modules, both those explicitly included using `<script type="module">` and those implicitly included using `import`, are loaded and executed in order. In the preceding example, the complete loading sequence is:

1. Download and parse `module1.js`.
2. Recursively download and parse `import` resources in `module1.js`.
3. Parse the inline module.
4. Recursively download and parse `import` resources in the inline module.
5. Download and parse `module2.js`.
6. Recursively download and parse `import` resources in `module2.js`

Once loading is complete, nothing is executed until after the document has been completely parsed. After document parsing completes, the following actions happen:

1. Recursively execute `import` resources for `module1.js`.
2. Execute `module1.js`.
3. Recursively execute `import` resources for the inline module.
4. Execute the inline module.
5. Recursively execute `import` resources for `module2.js`.
6. Execute `module2.js`.

Notice that the inline module acts like the other two modules except that the code doesn't have to be downloaded first. Otherwise, the sequence of loading `import` resources and executing modules is exactly the same.

I> The `defer` attribute is ignored on `<script type="module">` because it already behaves as if `defer` is applied.

Asynchronous Module Loading in Web Browsers

You may already be familiar with the `async` attribute on the `<script>` element. When used with scripts, `async` causes the script file to be executed as soon as the file is completely downloaded and parsed. The order of `async` scripts in the document doesn't affect the order in which the scripts are executed, though. The

scripts are always executed as soon as they finish downloading without waiting for the containing document to finish parsing.

The `async` attribute can be applied to modules as well. Using `async` on `<script type="module">` causes the module to execute in a manner similar to a script. The only difference is that all `import` resources for the module are downloaded before the module itself is executed. That guarantees all resources the module needs to function will be downloaded before the module executes; you just can't guarantee *when* the module will execute. Consider the following code:

```
<!-- no guarantee which one of these will execute first -->
<script type="module" async src="module1.js"></script>
<script type="module" async src="module2.js"></script>
```

In this example, there are two module files loaded asynchronously. It's not possible to tell which module will execute first simply by looking at this code. If `module1.js` finishes downloading first (including all of its `import` resources), then it will execute first. If `module2.js` finishes downloading first, then that module will execute first instead.

Loading Modules as Workers

Workers, such as web workers and service workers, execute JavaScript code outside of the web page context. Creating a new worker involves creating a new instance `Worker` (or another class) and passing in the location of JavaScript file. The default loading mechanism is to load files as scripts, like this:

```
// load script.js as a script
let worker = new Worker("script.js");
```

To support loading modules, the developers of the HTML standard added a second argument to these constructors. The second argument is an object with a `type` property with a default value of `"script"`. You can set `type` to `"module"` in order to load module files:

```
// load module.js as a module
let worker = new Worker("module.js", { type: "module" });
```

This example loads `module.js` as a module instead of a script by passing a second argument with `"module"` as the `type` property's value. (The `type` property is meant to mimic how the `type` attribute of `<script>` differentiates modules and scripts.) The second argument is supported for all worker types in the browser.

Worker modules are generally the same as worker scripts, but there are a couple of exceptions. First, worker scripts are limited to being loaded from the same origin as the web page in which they are referenced, but worker modules aren't quite as limited. Although worker modules have the same default restriction, they can also load files that have appropriate Cross-Origin Resource Sharing (CORS) headers to allow access. Second, while a worker script can use the `self.importScripts()` method to load additional scripts into the worker, `self.importScripts()` always fails on worker modules because you should use `import` instead.

Browser Module Specifier Resolution

All of the examples to this point in the chapter have used a relative module specifier path such as `"./example.js"`. Browsers require module specifiers to be in one of the following formats:

- Begin with `/` to resolve from the root directory
- Begin with `./` to resolve from the current directory
- Begin with `../` to resolve from the parent directory
- URL format

For example, suppose you have a module file located at `https://www.example.com/modules/module.js` that contains the following code:

```
// imports from https://www.example.com/modules/example1.js
import { first } from "./example1.js";

// imports from https://www.example.com/example2.js
import { second } from "../example2.js";

// imports from https://www.example.com/example3.js
import { third } from "/example3.js";

// imports from https://www2.example.com/example4.js
import { fourth } from "https://www2.example.com/example4.js";
```

Each of the module specifiers in this example is valid for use in a browser, including the complete URL in the final line (you'd need to be sure `ww2.example.com` has properly configured its Cross-Origin Resource Sharing (CORS) headers to allow cross-domain loading). These are the only module specifier formats that browsers can resolve by default (though the not-yet-complete module loader specification will provide ways to resolve other formats). That means some normal looking module specifiers are actually invalid in browsers and will result in an error, such as:

```
// invalid - doesn't begin with /, ./, or ../
import { first } from "example.js";

// invalid - doesn't begin with /, ./, or ../
import { second } from "example/index.js";
```

Each of these module specifiers cannot be loaded by the browser. The two module specifiers are in an invalid format (missing the correct beginning characters) even though both will work when used as the value of `src` in a `<script>` tag. This is an intentional difference in behavior between `<script>` and `import`.

Summary

ECMAScript 6 adds modules to the language as a way to package up and encapsulate functionality. Modules behave differently than scripts, as they don't modify the global scope with their top-level variables, functions, and classes, and `this` is `undefined`. To achieve that behavior, modules are loaded using a different mode.

You must export any functionality you'd like to make available to consumers of a module. Variables, functions, and classes can all be exported, and there is also one default export allowed per module. After exporting, another module can import all or some of the exported names. These names act as if defined by `let` and operate as block bindings that can't be redeclared in the same module.

Modules need not export anything if they are manipulating something in the global scope. You can actually import from such a module without introducing any bindings into the module scope.

Because modules must run in a different mode, browsers introduced `<script type="module">` to signal that the source file or inline code should be executed as a module. Module files loaded with `<script type="module">` are loaded as if the `defer` attribute is applied to them. Modules are also executed in the order in which they appear in the containing document once the document is fully parsed.