Chapter 10. Modules

The goal of modular programming is to allow large programs to be assembled using modules of code from disparate authors and sources and for all of that code to run correctly even in the presence of code that the various module authors did not anticipate. As a practical matter, modularity is mostly about encapsulating or hiding private implementation details and keeping the global namespace tidy so that modules cannot accidentally modify the variables, functions, and classes defined by other modules.

Until recently, JavaScript had no built-in support for modules, and programmers working on large code bases did their best to use the weak modularity available through classes, objects, and closures. Closure-based modularity, with support from code-bundling tools, led to a practical form of modularity based on a `require()` function, which was adopted by Node. `require()` -based modules are a fundamental part of the Node programming environment but were never adopted as an official part of the JavaScript language. Instead, ES6 defines modules using `import` and `export` keywords. Although `import` and `export` have been part of the language for years, they were only implemented by web browsers and Node relatively recently. And, as a practical matter, JavaScript modularity still depends on code-bundling tools.

The sections that follow cover:

- Do-it-yourself modules with classes, objects, and closures
- Node modules using `require()`
- ES6 modules using `export`, `import`, and `import()`

10.1 Modules with Classes, Objects, and Closures

Though it may be obvious, it is worth pointing out that one of the important features of classes is that they act as modules for their methods. Think back to Example 9-8. That example defined a number of different classes, all of which had a method named `has()`. But you would have no problem writing a program that used multiple set classes from that example: there is no danger that the implementation of `has()` from SingletonSet will overwrite the `has()` method of BitSet, for example.

The reason that the methods of one class are independent of the methods of other, unrelated classes is that the methods of each class are defined as properties of independent prototype objects. The reason that classes are modular is that objects are modular: defining a property in a JavaScript object is a lot like declaring a variable, but adding properties to objects does not affect the global namespace of a program, nor does it affect the properties of other objects. JavaScript defines quite a few mathematical functions and constants, but instead of defining them all globally, they are grouped as properties of a single global Math object. This same technique could have been used in Example 9-8. Instead of defining global classes with names like SingletonSet and BitSet, that example could have been written to define only a single global Sets object, with properties referencing the various classes. Users of this Sets library could then refer to the classes with names like `Sets.Singleton` and `Sets.Bit`.

Using classes and objects for modularity is a common and useful technique in JavaScript programming, but it doesn't go far enough. In particular, it doesn't offer us any way to hide internal implementation details inside the module. Consider Example 9-8 again. If we were writing that example as a module, maybe we would have wanted to keep the various abstract classes internal to the module, only making the concrete subclasses available to users of the module. Similarly, in the BitSet class, the `_valid()` and `_has()` methods are internal utilities that should not really be exposed to users of the class. And `BitSet.bits` and `BitSet.masks` are implementation details that would be better off hidden.

As we saw in §8.6, local variables and nested functions declared within a function are private to that function. This means that we can use immediately invoked function expressions to achieve a kind of modularity by leaving the implementation details and utility functions hidden within the enclosing function but making the public API of the module the return value of the function. In the case of the BitSet class, we might structure the module like this:

```
const BitSet = (function() { // Set BitSet to the return value of this func
    // Private implementation details here
    function isValid(set, n) { ... }
    function has(set, byte, bit) { ... }
    const BITS = new Uint8Array([1, 2, 4, 8, 16, 32, 64, 128]);
    const MASKS = new Uint8Array([~1, ~2, ~4, ~8, ~16, ~32, ~64, ~128]);

    // The public API of the module is just the BitSet class, which we defi
    // and return here. The class can use the private functions and constan
    // defined above, but they will be hidden from users of the class
    return class BitSet extends AbstractWritableSet {
        // ... implementation omitted ...
    };
}());
```

This approach to modularity becomes a little more interesting when the module has more than one item in it. The following code, for example, defines a mini statistics module that exports `mean()` and `stddev()` functions while leaving the implementation details hidden:

```
// This is how we could define a stats module
const stats = (function() {
    // Utility functions private to the module
    const sum = (x, y) => x + y;
    const square = x => x * x;

    // A public function that will be exported
    function mean(data) {
        return data.reduce(sum)/data.length;
    }

    // A public function that we will export
    function stddev(data) {
        let m = mean(data);
        return Math.sqrt(
            data.map(x => x - m).map(square).reduce(sum)/(data.length-1)
        );
    }

    // We export the public function as properties of an object
    return { mean, stddev };
}());

// And here is how we might use the module
stats.mean([1, 3, 5, 7, 9])   // => 5
stats.stddev([1, 3, 5, 7, 9]) // => Math.sqrt(10)
```

10.1.1 Automating Closure-Based Modularity

Note that it is a fairly mechanical process to transform a file of JavaScript code into this kind of module by inserting some text at the beginning and end of the file. All that is needed is some convention for the file of JavaScript code to indicate which values are to be exported and which are not.

Imagine a tool that takes a set of files, wraps the content of each of those files within an immediately invoked function expression, keeps track of the return value of each function, and concatenates everything into one big file. The result might look something like this:

```
const modules = {};
function require(moduleName) { return modules[moduleName]; }

modules["sets.js"] = (function() {
    const exports = {};

    // The contents of the sets.js file go here:
    exports.BitSet = class BitSet { ... };

    return exports;
}());

modules["stats.js"] = (function() {
    const exports = {};

    // The contents of the stats.js file go here:
    const sum = (x, y) => x + y;
    const square = x = > x * x;
    exports.mean = function(data) { ... };
    exports.stddev = function(data) { ... };

    return exports;
}());
```

With modules bundled up into a single file like the one shown in the preceding example, you can imagine writing code like the following to make use of those modules:

```
// Get references to the modules (or the module content) that we need
const stats = require("stats.js");
const BitSet = require("sets.js").BitSet;

// Now write code using those modules
let s = new BitSet(100);
s.insert(10);
s.insert(20);
s.insert(30);
let average = stats.mean([...s]); // average is 20
```

This code is a rough sketch of how code-bundling tools (such as webpack and Parcel) for web browsers work, and it's also a simple introduction to the `require()` function like the one used in Node programs.

10.2 Modules in Node

In Node programming, it is normal to split programs into as many files as seems natural. These files of JavaScript code are assumed to all live on a fast filesystem. Unlike web browsers, which have to read files of JavaScript over a relatively slow network connection, there is no need or benefit to bundling a Node program into a single JavaScript file.

In Node, each file is an independent module with a private namespace. Constants, variables, functions, and classes defined in one file are private to that file unless the file exports them. And values exported by one module are only visible in another module if that module explicitly imports them.

Node modules import other modules with the `require()` function and export their public API by setting properties of the Exports object or by replacing the `module.exports` object entirely.

10.2.1 Node Exports

Node defines a global `exports` object that is always defined. If you are writing a Node module that exports multiple values, you can simply assign them to the properties of this object:

```
const sum = (x, y) => x + y;
const square = x => x * x;

exports.mean = data => data.reduce(sum)/data.length;
exports.stddev = function(d) {
    let m = exports.mean(d);
    return Math.sqrt(d.map(x => x - m).map(square).reduce(sum)/(d.length-1)
};
```

Often, however, you want to define a module that exports only a single function or class rather than an object full of functions or classes. To do this, you simply assign the single value you want to export to `module.exports`:

```
module.exports = class BitSet extends AbstractWritableSet {
    // implementation omitted
};
```

The default value of `module.exports` is the same object that `exports` refers to. In the previous stats module, we could have assigned the mean function to `module.exports.mean` instead of `exports.mean`. Another approach with modules like the stats module is to export a single object at the end of the module rather than exporting functions one by one as you go:

```
// Define all the functions, public and private
const sum = (x, y) => x + y;
const square = x => x * x;
const mean = data => data.reduce(sum)/data.length;
const stddev = d => {
    let m = mean(d);
    return Math.sqrt(d.map(x => x - m).map(square).reduce(sum)/(d.length-1)
};

// Now export only the public ones
module.exports = { mean, stddev };
```

10.2.2 Node Imports

A Node module imports another module by calling the `require()` function. The argument to this function is the name of the module to be imported, and the return value is whatever value (typically a function, class, or object) that module exports.

If you want to import a system module built in to Node or a module that you have installed on your system via a package manager, then you simply use the unqualified name of the module, without any "/" characters that would turn it into a filesystem path:

```
// These modules are built in to Node
const fs = require("fs");           // The built-in filesystem module
const http = require("http");       // The built-in HTTP module

// The Express HTTP server framework is a third-party module.
// It is not part of Node but has been installed locally
const express = require("express");
```

When you want to import a module of your own code, the module name should be the path to the file that contains that code, relative to the current module's file. It is legal to use absolute paths that begin with a / character, but typically, when importing modules that are part of your own program, the module names will begin with ./ or sometimes ../ to indicate that they are relative to the current directory or the parent directory. For example:

```
const stats = require('./stats.js');
const BitSet = require('./utils/bitset.js');
```

(You can also omit the *.js* suffix on the files you're importing and Node will still find the files, but it is common to see these file extensions explicitly included.)

When a module exports just a single function or class, all you have to do is require it. When a module exports an object with multiple properties, you have a choice: you can import the entire object, or just import the specific properties (using destructuring assignment) of the object that you plan to use. Compare these two approaches:

```
// Import the entire stats object, with all of its functions
const stats = require('./stats.js');

// We've got more functions than we need, but they're neatly
// organized into a convenient "stats" namespace.
let average = stats.mean(data);

// Alternatively, we can use idiomatic destructuring assignment to import
// exactly the functions we want directly into the local namespace:
const { stddev } = require('./stats.js');

// This is nice and succinct, though we lose a bit of context
// without the 'stats' prefix as a namspace for the stddev() function.
let sd = stddev(data);
```

10.2.3 Node-Style Modules on the Web

Modules with an Exports object and a `require()` function are built in to Node. But if you're willing to process your code with a bundling tool like webpack, then it is also possible to use this style of modules for code that is intended to run in web browsers. Until recently, this was a very common thing to do, and you may see lots of web-based code that still does it.

Now that JavaScript has its own standard module syntax, however, developers who use bundlers are more likely to use the official JavaScript modules with `import` and `export` statements.

10.3 Modules in ES6

ES6 adds `import` and `export` keywords to JavaScript and finally supports real modularity as a core language feature. ES6 modularity is conceptually the same as Node modularity: each file is its own module, and constants, variables, functions, and classes defined within a file are private to that module unless they are explicitly exported. Values that are exported from one module are available for use in modules that explicitly import them. ES6 modules differ from Node modules in the syntax used for exporting and importing and also in the way that modules are defined in web browsers. The sections that follow explain these things in detail.

First, though, note that ES6 modules are also different from regular JavaScript "scripts" in some important ways. The most obvious difference is the modularity itself: in regular scripts, top-level declarations of variables, functions, and classes go into a single global context shared by all scripts. With modules, each file has its own private context and can use the `import` and `export` statements, which is the whole point, after all. But there are other differences between modules and scripts as well. Code inside an ES6 module (like code inside any ES6 `class` definition) is automatically in strict mode (see §5.6.3). This means that, when you start using ES6 modules, you'll never have to write `"use strict"` again. And it means that code in modules cannot use the `with` statement or the `arguments` object or undeclared variables. ES6 modules are even slightly stricter than strict mode: in strict mode, in functions invoked as functions, `this` is `undefined`. In modules, `this` is `undefined` even in top-level code. (By contrast, scripts in web browsers and Node set `this` to the global object.)

ES6 MODULES ON THE WEB AND IN NODE

ES6 modules have been in use on the web for years with the help of code bundlers like webpack, which combine independent modules of JavaScript code into large, non-modular bundles suitable for inclusion into web pages. At the time of this writing, however, ES6 modules are finally supported natively by all web browsers other than Internet Explorer. When used natively, ES6 modules are added into HTML pages with a special `<script type="module">` tag, described later in this chapter.

And meanwhile, having pioneered JavaScript modularity, Node finds itself in the awkward position of having to support two not entirely compatible module systems. Node 13 supports ES6 modules, but for now, the vast majority of Node programs still use Node modules.

10.3.1 ES6 Exports

To export a constant, variable, function, or class from an ES6 module, simply add the keyword `export` before the declaration:

```
export const PI = Math.PI;

export function degreesToRadians(d) { return d * PI / 180; }

export class Circle {
    constructor(r) { this.r = r; }
    area() { return PI * this.r * this.r; }
}
```

As an alternative to scattering `export` keywords throughout your module, you can define your constants, variables, functions, and classes as you normally would, with no `export` statement, and then (typically at the end of your module) write a single `export` statement that declares exactly what is exported in a single place. So instead of writing three individual exports in the preceding code, we could have equivalently written a single line at the end:

```
export { Circle, degreesToRadians, PI };
```

This syntax looks like the `export` keyword followed by an object literal (using shorthand notation). But in this case, the curly braces do not actually define an object literal. This export syntax simply requires a comma-separated list of identifiers within curly braces.

It is common to write modules that export only one value (typically a function or class), and in this case, we usually use `export default` instead of `export`:

```
export default class BitSet {
    // implementation omitted
}
```

Default exports are slightly easier to import than non-default exports, so when there is only one exported value, using `export default` makes things easier for the modules that use your exported value.

Regular exports with `export` can only be done on declarations that have a name. Default exports with `export default` can export any expression including anonymous function expressions and anonymous class expressions. This means that if you use `export default`, you can export object literals. So unlike the `export` syntax, if you see curly braces after `export default`, it really is an object literal that is being exported.

It is legal, but somewhat uncommon, for modules to have a set of regular exports and also a default export. If a module has a default export, it can only have one.

Finally, note that the `export` keyword can only appear at the top level of your JavaScript code. You may not export a value from within a class, function, loop, or conditional. (This is an important feature of the ES6 module system and enables static analysis: a modules export will be the same on every run, and the symbols exported can be determined before the module is actually run.)

10.3.2 ES6 Imports

You import values that have been exported by other modules with the `import` keyword. The simplest form of import is used for modules that define a default export:

```
import BitSet from './bitset.js';
```

This is the `import` keyword, followed by an identifier, followed by the `from` keyword, followed by a string literal that names the module whose default export we are importing. The default export value of the specified module becomes the value of the specified identifier in the current module.

The identifier to which the imported value is assigned is a constant, as if it had been declared with the `const` keyword. Like exports, imports can only appear at the top level of a module and are not allowed within classes, functions, loops, or conditionals. By near-universal convention, the imports needed by a module are placed at the start of the module. Interestingly, however, this is not required: like function declarations, imports are "hoisted" to the top, and all imported values are available for any of the module's code runs.

The module from which a value is imported is specified as a constant string literal in single quotes or double quotes. (You may not use a variable or other expression whose value is a string, and you may not use a string within backticks because template literals can interpolate variables and do not always have constant values.) In web browsers, this string is interpreted as a URL relative to the location of the module that is doing the importing. (In Node, or when using a bundling tool, the string is interpreted as a filename relative to the current module, but this makes little difference in practice.) A *module specifier* string must be an absolute path starting with "/", or a relative path starting with "./" or "../", or a complete URL a with protocol and hostname. The ES6 specification does not allow unqualified module specifier strings like "util.js" because it is ambiguous whether this is intended to name a module in the same directory as the current one or some kind of system module that is installed in some special location. (This restriction against "bare module specifiers" is not honored by code-bundling tools like webpack, which can easily be configured to find bare modules in a library directory that you specify.) A future version of the language may allow "bare module specifiers," but for now, they are not allowed. If you want to import a module from the same directory as the current one, simply place "./" before the module name and import from "./util.js" instead of "util.js".

So far, we've only considered the case of importing a single value from a module that uses `export default`. To import values from a module that exports multiple values, we use a slightly different syntax:

```
import { mean, stddev } from "./stats.js";
```

Recall that default exports do not need to have a name in the module that defines them. Instead, we provide a local name when we import those values. But non-default exports of a module do have names in the exporting module, and when we import those values, we refer to them by those names. The exporting module can export any number of named value. An `import` statement that references that module can import any subset of those values simply by listing their names within curly braces. The curly braces make this kind of `import` statement look something like a destructuring assignment, and destructuring assignment is actually a good analogy for what this style of import is doing. The identifiers within curly braces are all hoisted to the top of the importing module and behave like constants.

Style guides sometimes recommend that you explicitly import every symbol that your module will use. When importing from a module that defines many exports, however, you can easily import everything with an `import` statement like this:

```
import * as stats from "./stats.js";
```

An `import` statement like this creates an object and assigns it to a constant named `stats`. Each of the non-default exports of the module being imported becomes a property of this `stats` object. Non-default exports always have names, and those are used as property names within the object. Those properties are effectively constants: they cannot be overwritten or deleted. With the wildcard import shown in the previous example, the importing module would use the imported `mean()` and `stddev()` functions through the `stats` object, invoking them as `stats.mean()` and `stats.stddev()`.

Modules typically define either one default export or multiple named exports. It is legal, but somewhat uncommon, for a module to use both `export` and `export default`. But when a module does that, you can import both the default value and the named values with an `import` statement like this:

```
import Histogram, { mean, stddev } from "./histogram-stats.js";
```

So far, we've seen how to import from modules with a default export and from modules with non-default or named exports. But there is one other form of the `import` statement that is used with modules that have no exports at all. To include a no-exports module into your program, simply use the `import` keyword with the module specifier:

```
import "./analytics.js";
```

A module like this runs the first time it is imported. (And subsequent imports do nothing.) A module that just defines functions is only useful if it exports at least one of those functions. But if a module runs some code, then it can be useful to import even without symbols. An analytics module for a web application might run code to register various event handlers and then use those event handlers to send telemetry data back to the server at appropriate times. The module is self-contained and does not need to export anything, but we still need to `import` it so that it does actually run as part of our program.

Note that you can use this import-nothing `import` syntax even with modules that do have exports. If a module defines useful behavior independent of the values it exports, and if your program does not need any of those exported values, you can still import the module . just for that default behavior.

10.3.3 Imports and Exports with Renaming

If two modules export two different values using the same name and you want to import both of those values, you will have to rename one or both of the values when you import it. Similarly, if you want to import a value whose name is already in use in your module, you will need to rename the imported value. You can use the `as` keyword with named imports to rename them as you import them:

```
import { render as renderImage } from "./imageutils.js";
import { render as renderUI } from "./ui.js";
```

These lines import two functions into the current module. The functions are both named `render()` in the modules that define them but are imported with the more descriptive and disambiguating names `renderImage()` and `renderUI()`.

Recall that default exports do not have a name. The importing module always chooses the name when importing a default export. So there is no need for a special syntax for renaming in that case.

Having said that, however, the possibility of renaming on import provides another way of importing from modules that define both a default export and named exports. Recall the "./histogram-stats.js" module from the previous section. Here is another way to import both the default and named exports of that module:

```
import { default as Histogram, mean, stddev } from "./histogram-stats.js";
```

In this case, the JavaScript keyword `default` serves as a placeholder and allows us to indicate that we want to import and provide a name for the default export of the module.

It is also possible to rename values as you export them, but only when using the curly brace variant of the `export` statement. It is not common to need to do this, but if you chose short, succinct names for use inside your module, you might prefer to export your values with more descriptive names that are less likely to conflict with other modules. As with imports, you use the `as` keyword to do this:

```
export {
    layout as calculateLayout,
    render as renderLayout
};
```

Keep in mind that, although the curly braces look something like object literals, they are not, and the `export` keyword expects a single identifier before the `as`, not an expression. This means, unfortunately, that you cannot use export renaming like this:

```
export { Math.sin as sin, Math.cos as cos }; // SyntaxError
```

10.3.4 Re-Exports

Throughout this chapter, we've discussed a hypothetical "./stats.js" module that exports `mean()` and `stddev()` functions. If we were writing such a module and we thought that many users of the module would want only one function or the other, then we might want to define `mean()` in a "./stats/mean.js" module and define `stddev()` in "./stats/stddev.js". That way, programs only need to import exactly the functions they need and are not bloated by importing code they do not need.

Even if we had defined these statistical functions in individual modules, however, we might expect that there would be plenty of programs that want both functions and would appreciate a convenient "./stats.js" module from which they could import both on one line.

Given that the implementations are now in separate files, defining this "./stat.js" module is simple:

```
import { mean } from "./stats/mean.js";
import { stddev } from "./stats/stddev.js";
export { mean, stdev };
```

ES6 modules anticipate this use case and provide a special syntax for it. Instead of importing a symbol simply to export it again, you can combine the import and the export steps into a single "re-export" statement that uses the `export` keyword and the `from` keyword:

```
export { mean } from "./stats/mean.js";
export { stddev } from "./stats/stddev.js";
```

Note that the names `mean` and `stddev` are not actually used in this code. If we are not being selective with a re-export and simply want to export all of the named values from another module, we can use a wildcard:

```
export * from "./stats/mean.js";
export * from "./stats/stddev.js";
```

Re-export syntax allows renaming with `as` just as regular `import` and `export` statements do. Suppose we wanted to re-export the `mean()` function but also define `average()` as another name for the function. We could do that like this:

```
export { mean, mean as average } from "./stats/mean.js";
export { stddev } from "./stats/stddev.js";
```

All of the re-exports in this example assume that the “./stats/mean.js” and “./stats/stddev.js” modules export their functions using `export` instead of `export default`. In fact, however, since these are modules with only a single export, it would have made sense to define them with `export default`. If we had done so, then the re-export syntax is a little more complicated because it needs to define a name for the unnamed default exports. We can do that like this:

```
export { default as mean } from "./stats/mean.js";
export { default as stddev } from "./stats/stddev.js";
```

If you want to re-export a named symbol from another module as the default export of your module, you could do an `import` followed by an `export default`, or you could combine the two statements like this:

```
// Import the mean() function from ./stats.js and make it the
// default export of this module
export { mean as default } from "./stats.js"
```

And finally, to re-export the default export of another module as the default export of your module (though it is unclear why you would want to do this, since users could simply import the other module directly), you can write:

```
// The average.js module simply re-exports the stats/mean.js default export
export { default } from "./stats/mean.js"
```

10.3.5 JavaScript Modules on the Web

The preceding sections have described ES6 modules and their `import` and `export` declarations in a somewhat abstract manner. In this section and the next, we'll be discussing how they actually work in web browsers, and if you are not already an experienced web developer, you may find the rest of this chapter easier to understand after you have read Chapter 15.

As of early 2020, production code using ES6 modules is still generally bundled with a tool like webpack. There are trade-offs to doing this,[1] but on the whole, code bundling tends to give better performance. That may well change in the future as network speeds grow and browser vendors continue to optimize their ES6 module implementations.

Even though bundling tools may still be desirable in production, they are no longer required in development since all current browsers provide native support for JavaScript modules. Recall that modules use strict mode by default, `this` does not refer to a global object, and top-level declarations are not shared globally by default. Since modules must be executed differently than legacy non-module code, their introduction requires changes to HTML as well as JavaScript. If you want to natively use `import` directives in a web browser, you must tell the web browser that your code is a module by using a `<script type="module">` tag.

One of the nice features of ES6 modules is that each module has a static set of imports. So given a single starting module, a web browser can load all of its imported modules and then load all of the modules imported by that first batch of modules, and so on, until a complete program has been loaded. We've seen that the module specifier in an `import` statement can be treated as a relative URL. A `<script type="module">` tag marks the starting point of a modular program. None of the modules it imports are expected to be in `<script>` tags, however: instead, they are loaded on demand as regular JavaScript files and are executed in strict mode as regular ES6 modules. Using a `<script type="module">` tag to define the main entry point for a modular JavaScript program can be as simple as this:

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

Code inside an inline `<script type="module">` tag is an ES6 module, and as such can use the `export` statement. There is not any point in doing so, however, because the HTML `<script>` tag syntax does not provide any way to define a name for inline modules, so even if such a module does export a value, there is no way for another module to import it.

Scripts with the `type="module"` attribute are loaded and executed like scripts with the `defer` attribute. Loading of the code begins as soon as the HTML parser encounters the `<script>` tag (in the case of modules, this code-loading step may be a recursive process that loads multiple JavaScript files). But code execution does not begin until HTML parsing is complete. And once HTML parsing is complete, scripts (both modular and non) are executed in the order in which they appear in the HTML document.

You can modify the execution time of modules with the `async` attribute, which works the same way for modules that it does for regular scripts. An `async` module will execute as soon as the code is loaded, even if HTML parsing is not complete and even if this changes the relative ordering of the scripts.

Web browsers that support `<script type="module">` must also support `<script nomodule>`. Browsers that are module-aware ignore any script with the `nomodule` attribute and will not execute it. Browsers that do not support modules will not recognize the `nomodule` attribute, so they will ignore it and run the script. This provides a powerful technique for dealing with browser compatibility issues. Browsers that support ES6 modules also support other modern JavaScript features like classes, arrow functions, and the `for/of` loop. If you write modern JavaScript and load it with `<script type="module">`, you know that it will only be loaded by browsers that can support it. And as a fallback for IE11 (which, in 2020, is effectively the only remaining browser that does not support ES6), you can use tools like Babel and webpack to transform your code into non-modular ES5 code, then load that less-efficient transformed code via `<script nomodule>`.

Another important difference between regular scripts and module scripts has to do with cross-origin loading. A regular `<script>` tag will load a file of JavaScript code from any server on the internet, and the internet's infrastructure of advertising, analytics, and tracking code depends on that fact. But `<script type="module">` provides an opportunity to tighten this up, and modules can only be loaded from the same origin as the containing HTML document or when proper CORS headers are in place to securely allow cross-origin loads. An unfortunate side effect of this new security restriction is that it makes it difficult to test ES6 modules in development mode using `file:` URLs. When using ES6 modules, you will likely need to set up a static web server for testing.

Some programmers like to use the filename extension `.mjs` to distinguish their modular JavaScript files from their regular, non-modular JavaScript files with the traditional `.js` extension. For the purposes of web browsers and `<script>` tags, the file extension is actually irrelevant. (The MIME type is relevant, however, so if you use `.mjs` files, you may need to configure your web server to serve them with the same MIME type as `.js` files.) Node's support for ES6 does use the filename extension as a hint to distinguish which module system is used by each file it loads. So if you are writing ES6 modules and want them to be usable with Node, then it may be helpful to adopt the `.mjs` naming convention.

10.3.6 Dynamic Imports with import()

We've seen that the ES6 `import` and `export` directives are completely static and enable JavaScript interpreters and other JavaScript tools to determine the relationships between modules with simple text analysis while the modules are being loaded without having to actually execute any of the code in the modules. With statically imported modules, you are guaranteed that the values you import into a module will be ready for use before any of the code in your module begins to run.

On the web, code has to be transferred over a network instead of being read from the filesystem. And once transfered, that code is often executed on mobile devices with relatively slow CPUs. This is not the kind of environment where static module imports—which require an entire program to be loaded before any of it runs—make a lot of sense.

It is common for web applications to initially load only enough of their code to render the first page displayed to the user. Then, once the user has some preliminary content to interact with, they can begin to load the often much larger amount of code needed for the rest of the web app. Web browsers make it easy to dynamically load code by using the DOM API to inject a new `<script>` tag into the current HTML document, and web apps have been doing this for many years.

Although dynamic loading has been possible for a long time, it has not been part of the language itself. That changes with the introduction of `import()` in ES2020 (as of early 2020, dynamic import is supported by all browsers that support ES6 modules). You pass a module specifier to `import()` and it returns a Promise object that represents the asynchronous process of loading and running the specified module. When the dynamic import is complete, the Promise is "fulfilled" (see Chapter 13 for complete details on asynchronous programming and Promises) and produces an object like the one you would get with the `import * as` form of the static import statement.

So instead of importing the "./stats.js" module statically, like this:

```
import * as stats from "./stats.js";
```

we might import it and use it dynamically, like this:

```
import("./stats.js").then(stats => {
    let average = stats.mean(data);
})
```

Or, in an `async` function (again, you may need to read Chapter 13 before you'll understand this code), we can simplify the code with `await`:

```
async analyzeData(data) {
    let stats = await import("./stats.js");
    return {
        average: stats.mean(data),
        stddev: stats.stddev(data)
    };
}
```

The argument to `import()` should be a module specifier, exactly like one you'd use with a static `import` directive. But with `import()`, you are not constrained to use a constant string literal: any expression that evaluates to a string in the proper form will do.

Dynamic `import()` looks like a function invocation, but it actually is not. Instead, `import()` is an operator and the parentheses are a required part of the operator syntax. The reason for this unusual bit of syntax is that `import()` needs to be able to resolve module specifiers as URLs relative to the currently running module, and this requires a bit of implementation magic that would not be legal to put in a JavaScript function. The function versus operator distinction rarely makes a difference in practice, but you'll notice it if you try writing code like `console.log(import);` or `let require = import;`.

Finally, note that dynamic `import()` is not just for web browsers. Code-packaging tools like webpack can also make good use of it. The most straightforward way to use a code bundler is to tell it the main entry point for your program and let it find all the static `import` directives and assemble everything into one large file. By strategically using dynamic `import()` calls, however, you can break that one monolithic bundle up into a set of smaller bundles that can be loaded on demand.

10.3.7 import.meta.url

There is one final feature of the ES6 module system to discuss. Within an ES6 module (but not within a regular `<script>` or a Node module loaded with `require()`), the special syntax `import.meta` refers to an object that contains metadata about the currently executing module. The `url` property of this object is the URL from which the module was loaded. (In Node, this will be a `file://` URL.)

The primary use case of `import.meta.url` is to be able to refer to images, data files, or other resources that are stored in the same directory as (or relative to) the module. The `URL()` constructor makes it easy to resolve a relative URL against an absolute URL like `import.meta.url`. Suppose, for example, that you have written a module that includes strings that need to be localized and that the localization files are stored in an `l10n/` directory, which is in the same directory as the module itself. Your module could load its strings using a URL created with a function, like this:

```
function localStringsURL(locale) {
    return new URL(`l10n/${locale}.json`, import.meta.url);
}
```

10.4 Summary

The goal of modularity is to allow programmers to hide the implementation details of their code so that chunks of code from various sources can be assembled into large programs without worrying that one chunk will overwrite functions or variables of another. This chapter has explained three different JavaScript module systems:

- In the early days of JavaScript, modularity could only be achieved through the clever use of immediately invoked function expressions.
- Node added its own module system on top of the JavaScript language. Node modules are imported with `require()` and define their exports by setting properties of the Exports object, or by setting the `module.exports` property.
- In ES6, JavaScript finally got its own module system with `import` and `export` keywords, and ES2020 is adding support for dynamic imports with `import()`.

[1] For example: web apps that have frequent incremental updates and users who make frequent return visits may find that using small modules instead of large bundles can result in better average load times because of better utilization of the user's browser cache.