You Don't Know JS Yet: Scope & Closures - 2nd Edition

Chapter 8: The Module Pattern

In this chapter, we wrap up the main text of the book by exploring one of the most important code organization patterns in all of programming: the module. As we'll see, modules are inherently built from what we've already covered: the payoff for your efforts in learning lexical scope and closure.

We've examined every angle of lexical scope, from the breadth of the global scope down through nested block scopes, into the intricacies of the variable lifecycle. Then we leveraged lexical scope to understand the full power of closure.

Take a moment to reflect on how far you've come in this journey so far; you've taken big steps in getting to know JS more deeply!

The central theme of this book has been that understanding and mastering scope and closure is key in properly structuring and organizing our code, especially the decisions on where to store information in variables.

Our goal in this final chapter is to appreciate how modules embody the importance of these topics, elevating them from abstract concepts to concrete, practical improvements in building programs.

Encapsulation and Least Exposure (POLE)

Encapsulation is often cited as a principle of object-oriented (OO) programming, but it's more fundamental and broadly applicable than that. The goal of encapsulation is the bundling or co-location of information (data) and behavior (functions) that together serve a common purpose.

Independent of any syntax or code mechanisms, the spirit of encapsulation can be realized in something as simple as using separate files to hold bits of the overall program with common purpose. If we bundle everything that powers a list of search results into a single file called "search-list.js", we're encapsulating that part of the program.

The recent trend in modern front-end programming to organize applications around Component architecture pushes encapsulation even further. For many, it feels natural to consolidate everything that constitutes the search results list—even beyond code, including presentational markup and styling—into a single unit of program logic, something tangible we can interact with. And then we label that collection the "SearchList" component.

Another key goal is the control of visibility of certain aspects of the encapsulated data and functionality. Recall from Chapter 6 the *least exposure* principle (POLE), which seeks to defensively guard against various *dangers* of scope over-exposure; these affect both variables and functions. In JS, we most often implement visibility control through the mechanics of lexical scope.

The idea is to group alike program bits together, and selectively limit programmatic access to the parts we consider *private* details. What's not considered *private* is then marked as *public*, accessible to the whole program.

The natural effect of this effort is better code organization. It's easier to build and maintain software when we know where things are, with clear and obvious boundaries and connection points. It's also easier to maintain quality if we avoid the pitfalls of over-exposed data and functionality.

These are some of the main benefits of organizing JS programs into modules.

What Is a Module?

A module is a collection of related data and functions (often referred to as methods in this context), characterized by a division between hidden *private* details and *public* accessible details, usually called the "public API."

A module is also stateful: it maintains some information over time, along with functionality to access and update that information.

> **NOTE:**
>
> A broader concern of the module pattern is fully embracing system-level modularization through loose-coupling and other program architecture techniques. That's a complex topic well beyond the bounds of our discussion, but is worth further study beyond this book.

To get a better sense of what a module is, let's compare some module characteristics to useful code patterns that aren't quite modules.

Namespaces (Stateless Grouping)

If you group a set of related functions together, without data, then you don't really have the expected encapsulation a module implies. The better term for this grouping of *stateless* functions is a namespace:

```
// namespace, not module
var Utils = {
    cancelEvt(evt) {
        evt.preventDefault();
        evt.stopPropagation();
        evt.stopImmediatePropagation();
    },
    wait(ms) {
        return new Promise(function c(res){
            setTimeout(res,ms);
        });
    },
    isValidEmail(email) {
        return /[^@]+@[^@.]+\.[^@.]+/.test(email);
    }
};
```

`Utils` here is a useful collection of utilities, yet they're all state-independent functions. Gathering functionality together is generally good practice, but that doesn't make this a module. Rather, we've defined a `Utils` namespace and organized the functions under it.

Data Structures (Stateful Grouping)

Even if you bundle data and stateful functions together, if you're not limiting the visibility of any of it, then you're stopping short of the POLE aspect of encapsulation; it's not particularly helpful to label that a module.

Consider:

```
// data structure, not module
var Student = {
    records: [
        { id: 14, name: "Kyle", grade: 86 },
        { id: 73, name: "Suzy", grade: 87 },
        { id: 112, name: "Frank", grade: 75 },
        { id: 6, name: "Sarah", grade: 91 }
    ],
    getName(studentID) {
        var student = this.records.find(
            student => student.id == studentID
        );
        return student.name;
    }
};

Student.getName(73);
// Suzy
```

Since `records` is publicly accessible data, not hidden behind a public API, `Student` here isn't really a module.

`Student` does have the data-and-functionality aspect of encapsulation, but not the visibility-control aspect. It's best to label this an instance of a data structure.

Modules (Stateful Access Control)

To embody the full spirit of the module pattern, we not only need grouping and state, but also access control through visibility (private vs. public).

Let's turn `Student` from the previous section into a module. We'll start with a form I call the "classic module," which was originally referred to as the "revealing module" when it first emerged in the early 2000s. Consider:

```
var Student = (function defineStudent(){
    var records = [
        { id: 14, name: "Kyle", grade: 86 },
        { id: 73, name: "Suzy", grade: 87 },
        { id: 112, name: "Frank", grade: 75 },
        { id: 6, name: "Sarah", grade: 91 }
    ];

    var publicAPI = {
        getName
    };

    return publicAPI;

    // ************************

    function getName(studentID) {
        var student = records.find(
            student => student.id == studentID
        );
        return student.name;
    }
})();

Student.getName(73);   // Suzy
```

`Student` is now an instance of a module. It features a public API with a single method: `getName(..)`. This method is able to access the private hidden `records` data.

WARNING:

I should point out that the explicit student data being hard-coded into this module definition is just for our illustration purposes. A typical module in your program will receive this data from an outside source, typically loaded from databases, JSON data files, Ajax calls, etc. The data is then injected into the module instance typically through method(s) on the module's public API.

How does the classic module format work?

Notice that the instance of the module is created by the `defineStudent()` IIFE being executed. This IIFE returns an object (named `publicAPI`) that has a property on it referencing the inner `getName(..)` function.

Naming the object `publicAPI` is stylistic preference on my part. The object can be named whatever you like (JS doesn't care), or you can just return an object directly without assigning it to any internal named variable. More on this choice in Appendix A.

From the outside, `Student.getName(..)` invokes this exposed inner function, which maintains access to the inner `records` variable via closure.

You don't *have* to return an object with a function as one of its properties. You could just return a function directly, in place of the object. That still satisfies all the core bits of a classic module.

By virtue of how lexical scope works, defining variables and functions inside your outer module definition function makes everything *by default* private. Only properties added to the public API object returned from the function will be exported for external public use.

The use of an IIFE implies that our program only ever needs a single central instance of the module, commonly referred to as a "singleton." Indeed, this specific example is simple enough that there's no obvious reason we'd need anything more than just one instance of the `Student` module.

Module Factory (Multiple Instances)

But if we did want to define a module that supported multiple instances in our program, we can slightly tweak the code:

```
// factory function, not singleton IIFE
function defineStudent() {
    var records = [
        { id: 14, name: "Kyle", grade: 86 },
        { id: 73, name: "Suzy", grade: 87 },
        { id: 112, name: "Frank", grade: 75 },
        { id: 6, name: "Sarah", grade: 91 }
    ];

    var publicAPI = {
        getName
    };

    return publicAPI;

    // ************************

    function getName(studentID) {
        var student = records.find(
            student => student.id == studentID
        );
        return student.name;
    }
}

var fullTime = defineStudent();
fullTime.getName(73);           // Suzy
```

Rather than specifying `defineStudent()` as an IIFE, we just define it as a normal standalone function, which is commonly referred to in this context as a "module factory" function.

We then call the module factory, producing an instance of the module that we label `fullTime`. This module instance implies a new instance of the inner scope, and thus a new closure that `getName(..)` holds over `records`. `fullTime.getName(..)` now invokes the method on that specific instance.

Classic Module Definition

So to clarify what makes something a classic module:

- There must be an outer scope, typically from a module factory function running at least once.
- The module's inner scope must have at least one piece of hidden information that represents state for the module.
- The module must return on its public API a reference to at least one function that has closure over the hidden module state (so that this state is actually preserved).

You'll likely run across other variations on this classic module approach, which we'll look at in more detail in Appendix A.

Node CommonJS Modules

In Chapter 4, we introduced the CommonJS module format used by Node. Unlike the classic module format described earlier, where you could bundle the module factory or IIFE alongside any other code including other modules, CommonJS modules are file-based; one module per file.

Let's tweak our module example to adhere to that format:

```
module.exports.getName = getName;

// ***********************

var records = [
    { id: 14, name: "Kyle", grade: 86 },
    { id: 73, name: "Suzy", grade: 87 },
    { id: 112, name: "Frank", grade: 75 },
    { id: 6, name: "Sarah", grade: 91 }
];

function getName(studentID) {
    var student = records.find(
        student => student.id == studentID
    );
    return student.name;
}
```

The `records` and `getName` identifiers are in the top-level scope of this module, but that's not the global scope (as explained in Chapter 4). As such, everything here is *by default* private to the module.

To expose something on the public API of a CommonJS module, you add a property to the empty object provided as `module.exports`. In some older legacy code, you may run across references to just a bare `exports`, but for code clarity you should always fully qualify that reference with the `module.` prefix.

For style purposes, I like to put my "exports" at the top and my module implementation at the bottom. But these exports can be placed anywhere. I strongly recommend collecting them all together, either at the top or bottom of your file.

Some developers have the habit of replacing the default exports object, like this:

```
// defining a new object for the API
module.exports = {
    // ..exports..
};
```

There are some quirks with this approach, including unexpected behavior if multiple such modules circularly depend on each other. As such, I recommend against replacing the object. If you want to assign multiple exports at once, using object literal style definition, you can do this instead:

```
Object.assign(module.exports,{
   // .. exports ..
});
```

What's happening here is defining the `{ .. }` object literal with your module's public API specified, and then `Object.assign(..)` is performing a shallow copy of all those properties onto the existing `module.exports` object, instead of replacing it This is a nice balance of convenience and safer module behavior.

To include another module instance into your module/program, use Node's `require(..)` method. Assuming this module is located at "/path/to/student.js", this is how we can access it:

```
var Student = require("/path/to/student.js");

Student.getName(73);
// Suzy
```

`Student` now references the public API of our example module.

CommonJS modules behave as singleton instances, similar to the IIFE module definition style presented before. No matter how many times you `require(..)` the same module, you just get additional references to the single shared module instance.

`require(..)` is an all-or-nothing mechanism; it includes a reference of the entire exposed public API of the module. To effectively access only part of the API, the typical approach looks like this:

```
var getName = require("/path/to/student.js").getName;

// or alternately:

var { getName } = require("/path/to/student.js");
```

Similar to the classic module format, the publicly exported methods of a CommonJS module's API hold closures over the internal module details. That's how the module singleton state is maintained across the lifetime of your program.

NOTE:

In Node `require("student")` statements, non-absolute paths (`"student"`) assume a ".js" file extension and search "node_modules".

Modern ES Modules (ESM)

The ESM format shares several similarities with the CommonJS format. ESM is file-based, and module instances are singletons, with everything private *by default*. One notable difference is that ESM files are assumed to be strict-mode, without needing a `"use strict"` pragma at the top. There's no way to define an ESM as non-strict-mode.

Instead of `module.exports` in CommonJS, ESM uses an `export` keyword to expose something on the public API of the module. The `import` keyword replaces the `require(..)` statement. Let's adjust "students.js" to use the ESM format:

```
export { getName };

// ***********************

var records = [
    { id: 14, name: "Kyle", grade: 86 },
    { id: 73, name: "Suzy", grade: 87 },
    { id: 112, name: "Frank", grade: 75 },
    { id: 6, name: "Sarah", grade: 91 }
];

function getName(studentID) {
    var student = records.find(
        student => student.id == studentID
    );
    return student.name;
}
```

The only change here is the `export { getName }` statement. As before, `export` statements can appear anywhere throughout the file, though `export` must be at the top-level scope; it cannot be inside any other block or function.

ESM offers a fair bit of variation on how the `export` statements can be specified. For example:

```
export function getName(studentID) {
    // ..
}
```

Even though `export` appears before the `function` keyword here, this form is still a `function` declaration that also happens to be exported. That is, the `getName` identifier is *function hoisted* (see Chapter 5), so it's available throughout the whole scope of the module.

Another allowed variation:

```
export default function getName(studentID) {
    // ..
}
```

This is a so-called "default export," which has different semantics from other exports. In essence, a "default export" is a shorthand for consumers of the module when they `import`, giving them a terser syntax when they only need this single default API member.

Non-`default` exports are referred to as "named exports."

The `import` keyword—like `export`, it must be used only at the top level of an ESM outside of any blocks or functions—also has a number of variations in syntax. The first is referred to as "named import":

```
import { getName } from "/path/to/students.js";

getName(73);   // Suzy
```

As you can see, this form imports only the specifically named public API members from a module (skipping anything not named explicitly), and it adds those identifiers to the top-level scope of the current module. This type of import is a familiar style to those used to package imports in languages like Java.

Multiple API members can be listed inside the `{ .. }` set, separated with commas. A named import can also be *renamed* with the `as` keyword:

```
import { getName as getStudentName }
   from "/path/to/students.js";

getStudentName(73);
// Suzy
```

If `getName` is a "default export" of the module, we can import it like this:

```
import getName from "/path/to/students.js";

getName(73);   // Suzy
```

The only difference here is dropping the `{ }` around the import binding. If you want to mix a default import with other named imports:

```
import { default as getName, /* .. others .. */ }
   from "/path/to/students.js";

getName(73);   // Suzy
```

By contrast, the other major variation on `import` is called "namespace import":

```
import * as Student from "/path/to/students.js";

Student.getName(73);   // Suzy
```

As is likely obvious, the `*` imports everything exported to the API, default and named, and stores it all under the single namespace identifier as specified. This approach most closely matches the form of classic modules for most of JS's history.

NOTE:

As of the time of this writing, modern browsers have supported ESM for a few years now, but Node's stable'ish support for ESM is fairly recent, and has been evolving for quite a while. The evolution is likely to continue for another year or more; the introduction of ESM to JS back in ES6 created a number of challenging compatibility concerns for Node's interop with CommonJS modules. Consult Node's ESM documentation for all the latest details: https://nodejs.org/api/esm.html

Exit Scope

Whether you use the classic module format (browser or Node), CommonJS format (in Node), or ESM format (browser or Node), modules are one of the most effective ways to structure and organize your program's functionality and data.

The module pattern is the conclusion of our journey in this book of learning how we can use the rules of lexical scope to place variables and functions in proper locations. POLE is the defensive *private by default* posture we always take, making sure we avoid over-exposure and interact only with the minimal public API surface area necessary.

And underneath modules, the *magic* of how all our module state is maintained is closures leveraging the lexical scope system.

That's it for the main text. Congratulations on quite a journey so far! As I've said numerous times throughout, it's a really good idea to pause, reflect, and practice what we've just discussed.

When you're comfortable and ready, check out the appendices, which dig deeper into some of the corners of these topics, and also challenge you with some practice exercises to solidify what you've learned.