MODULES

> "Write code that is easy to delete, not easy to extend."
>
> —Tef, *Programming is Terrible*

Ideally, a program has a clear, straightforward structure. The way it works is easy to explain, and each part plays a well-defined role.

In practice, programs grow organically. Pieces of functionality are added as the programmer identifies new needs. Keeping such a program well-structured requires constant attention and work. This is work that will pay off only in the future, the *next* time someone works on the program, so it's tempting to neglect it and allow the various parts of the program to become deeply entangled.

This causes two practical issues. First, understanding an entangled system is hard. If everything can touch everything else, it is difficult to look at any given piece in isolation. You are forced to build up a holistic understanding of the entire thing. Second, if you want to use any of the functionality from such a program in another situation, rewriting it may be easier than trying to disentangle it from its context.

The phrase “big ball of mud” is often used for such large, structureless programs. Everything sticks together, and when you try to pick out a piece, the whole thing comes apart, and you only succeed in making a mess.

Modular programs

Modules are an attempt to avoid these problems. A module is a piece of program that specifies which other pieces it relies on and which functionality it provides for other modules to use (its *interface*).

Module interfaces have a lot in common with object interfaces, as we saw them in Chapter 6. They make part of the module available to the outside world and keep the rest private.

But the interface that a module provides for others to use is only half the story. A good module system also requires modules to specify which code *they* use from other modules. These relations are called *dependencies*. If module A uses functionality from module B, it is said to *depend* on that module. When these are clearly specified in the module itself, they can be used to figure out which other modules need to be present to be able to use a given module and to automatically load dependencies.

When the ways in which modules interact with each other are explicit, a system becomes more like LEGO, where pieces interact through well-defined connectors, and less like mud, where everything mixes with everything else.

ES modules

The original JavaScript language did not have any concept of a module. All scripts ran in the same scope, and accessing a function defined in another script was done by referencing the global bindings created by that script. This actively encouraged accidental, hard-to-see entanglement of code and invited problems like unrelated scripts trying to use the same binding name.

Since ECMAScript 2015, JavaScript supports two different types of programs. *Scripts* behave in the old way: their bindings are defined in the global scope, and they have no way to directly reference other scripts. *Modules* get their own separate scope and support the `import` and `export` keywords, which

aren't available in scripts, to declare their dependencies and interface. This module system is usually called *ES modules* (where *ES* stands for ECMAScript).

A modular program is composed of a number of such modules, wired together via their imports and exports.

The following example module converts between day names and numbers (as returned by `Date`'s `getDay` method). It defines a constant that is not part of its interface, and two functions that are. It has no dependencies.

```
const names = ["Sunday", "Monday", "Tuesday", "Wednesday",
               "Thursday", "Friday", "Saturday"];

export function dayName(number) {
  return names[number];
}
export function dayNumber(name) {
  return names.indexOf(name);
}
```

The `export` keyword can be put in front of a function, class, or binding definition to indicate that that binding is part of the module's interface. This makes it possible for other modules to use that binding by importing it.

```
import {dayName} from "./dayname.js";
let now = new Date();
console.log(`Today is ${dayName(now.getDay())}`);
// → Today is Monday
```

The `import` keyword, followed by a list of binding names in braces, makes bindings from another module available in the current module. Modules are identified by quoted strings.

How such a module name is resolved to an actual program differs by platform. The browser treats them as web addresses, whereas Node.js resolves them to files. When you run a module, all the other modules it depends on—and the modules *those* depend on—are loaded, and the exported bindings are made available to the modules that import them.

Import and export declarations cannot appear inside of functions, loops, or other blocks. They are immediately resolved when the module is loaded, regardless of how the code in the module executes. To reflect this, they must appear only in the outer module body.

A module's interface thus consists of a collection of named bindings, which other modules that depend on the module can access. Imported bindings can be renamed to give them a new local name using `as` after their name.

```
import {dayName as nomDeJour} from "./dayname.js";
console.log(nomDeJour(3));
// → Wednesday
```

A module may also have a special export named `default`, which is often used for modules that only export a single binding. To define a default export, you write `export default` before an expression, a function declaration, or a class declaration.

```
export default ["Winter", "Spring", "Summer", "Autumn"];
```

Such a binding is imported by omitting the braces around the name of the import.

```
import seasonNames from "./seasonname.js";
```

To import all bindings from a module at the same time, you can use `import *`. You provide a name, and that name will be bound to an object holding all the module's exports. This can be useful when you are using a lot of different exports.

```
import * as dayName from "./dayname.js";
console.log(dayName.dayName(3));
// → Wednesday
```

Packages

One of the advantages of building a program out of separate pieces and being able to run some of those pieces on their own is that you might be able to use the same piece in different programs.

But how do you set this up? Say I want to use the `parseINI` function from Chapter 9 in another program. If it is clear what the function depends on (in this case, nothing), I can just copy that module into my new project and use it. But then, if I find a mistake in the code, I'll probably fix it in whichever program I'm working with at the time and forget to also fix it in the other program.

Once you start duplicating code, you'll quickly find yourself wasting time and energy moving copies around and keeping them up to date. That's where *packages* come in. A package is a chunk of code that can be distributed (copied and installed). It may contain one or more modules and has information about which other packages it depends on. A package also usually comes with documentation explaining what it does so that people who didn't write it might still be able to use it.

When a problem is found in a package or a new feature is added, the package is updated. Now the programs that depend on it (which may also be packages) can copy the new version to get the improvements that were made to the code.

Working in this way requires infrastructure. We need a place to store and find packages and a convenient way to install and upgrade them. In the JavaScript world, this infrastructure is provided by NPM (*https://npmjs.org*).

NPM is two things: an online service where you can download (and upload) packages, and a program (bundled with Node.js) that helps you install and manage them.

At the time of writing, there are more than three million different packages available on NPM. A large portion of those are rubbish, to be fair. But almost every useful, publicly available JavaScript package can be found on NPM. For example, an INI file parser, similar to the one we built in Chapter 9, is available under the package name `ini`.

Chapter 20 will show how to install such packages locally using the `npm` command line program.

Having quality packages available for download is extremely valuable. It means that we can often avoid reinventing a program that 100 people have

written before and get a solid, well-tested implementation at the press of a few keys.

Software is cheap to copy, so once someone has written it, distributing it to other people is an efficient process. Writing it in the first place *is* work, though, and responding to people who have found problems in the code or who want to propose new features is even more work.

By default, you own the copyright to the code you write, and other people may use it only with your permission. But because some people are just nice and because publishing good software can help make you a little bit famous among programmers, many packages are published under a license that explicitly allows other people to use it.

Most code on NPM is licensed this way. Some licenses require you to also publish code that you build on top of the package under the same license. Others are less demanding, requiring only that you keep the license with the code as you distribute it. The JavaScript community mostly uses the latter type of license. When using other people's packages, make sure you are aware of their licenses.

Now, instead of writing our own INI file parser, we can use one from NPM.

```
import {parse} from "ini";

console.log(parse("x = 10\ny = 20"));
// → {x: "10", y: "20"}
```

CommonJS modules

Before 2015, when the JavaScript language had no built-in module system, people were already building large systems in JavaScript. To make that workable, they *needed* modules.

The community designed its own improvised module systems on top of the language. These use functions to create a local scope for the modules and regular objects to represent module interfaces.

Initially, people just manually wrapped their entire module in an "immediately invoked function expression" to create the module's scope and assigned their interface objects to a single global variable.

```
const weekDay = function() {
  const names = ["Sunday", "Monday", "Tuesday", "Wednesday",
                 "Thursday", "Friday", "Saturday"];
  return {
    name(number) { return names[number]; },
    number(name) { return names.indexOf(name); }
  };
}();

console.log(weekDay.name(weekDay.number("Sunday")));
// → Sunday
```

This style of modules provides isolation, to a certain degree, but it does not declare dependencies. Instead, it just puts its interface into the global scope and expects its dependencies, if any, to do the same. This is not ideal.

If we implement our own module loader, we can do better. The most widely used approach to bolted-on JavaScript modules is called *CommonJS modules*. Node.js used this module system from the start (though it now also knows how to load ES modules), and it is the module system used by many packages on NPM.

A CommonJS module looks like a regular script, but it has access to two bindings that it uses to interact with other modules. The first is a function called `require`. When you call this with the module name of your dependency, it makes sure the module is loaded and returns its interface. The second is an object named `exports`, which is the interface object for the module. It starts out empty and you add properties to it to define exported values.

This CommonJS example module provides a date-formatting function. It uses two packages from NPM—`ordinal` to convert numbers to strings like `"1st"` and `"2nd"`, and `date-names` to get the English names for weekdays and months. It exports a single function, `formatDate`, which takes a `Date` object and a template string.

The template string may contain codes that direct the format, such as `YYYY` for the full year and `Do` for the ordinal day of the month. You could give it a string like `"MMMM Do YYYY"` to get output like `November 22nd 2017`.

```
const ordinal = require("ordinal");
const {days, months} = require("date-names");

exports.formatDate = function(date, format) {
  return format.replace(/YYYY|M(MMM)?|Do?|dddd/g, tag => {
    if (tag == "YYYY") return date.getFullYear();
    if (tag == "M") return date.getMonth();
    if (tag == "MMMM") return months[date.getMonth()];
    if (tag == "D") return date.getDate();
    if (tag == "Do") return ordinal(date.getDate());
    if (tag == "dddd") return days[date.getDay()];
  });
};
```

The interface of `ordinal` is a single function, whereas `date-names` exports an object containing multiple things—`days` and `months` are arrays of names. Destructuring is very convenient when creating bindings for imported interfaces.

The module adds its interface function to `exports` so that modules that depend on it get access to it. We could use the module like this:

```
const {formatDate} = require("./format-date.js");

console.log(formatDate(new Date(2017, 9, 13),
                       "dddd the Do"));
// → Friday the 13th
```

CommonJS is implemented with a module loader that, when loading a module, wraps its code in a function (giving it its own local scope), and passes the `require` and `exports` bindings to that function as arguments.

If we assume we have access to a `readFile` function that reads a file by name and gives us its content, we can define a simplified form of `require` like this:

```
function require(name) {
  if (!(name in require.cache)) {
```

```
      let code = readFile(name);
      let exports = require.cache[name] = {};
      let wrapper = Function("require, exports", code);
      wrapper(require, exports);
    }
    return require.cache[name];
  }
  require.cache = Object.create(null);
```

`Function` is a built-in JavaScript function that takes a list of arguments (as a comma-separated string) and a string containing the function body and returns a function value with those arguments and that body. This is an interesting concept—it allows a program to create new pieces of program from string data—but also a dangerous one, since if someone can trick your program into putting a string they provide into `Function`, they can make the program do anything they want.

Standard JavaScript provides no such function as `readFile`, but different JavaScript environments, such as the browser and Node.js, provide their own ways of accessing files. The example just pretends that `readFile` exists.

To avoid loading the same module multiple times, `require` keeps a store (cache) of already loaded modules. When called, it first checks if the requested module has been loaded and, if not, loads it. This involves reading the module's code, wrapping it in a function, and calling it.

By defining `require`, `exports` as parameters for the generated wrapper function (and passing the appropriate values when calling it), the loader makes sure that these bindings are available in the module's scope.

An important difference between this system and ES modules is that ES module imports happen before a module's script starts running, whereas `require` is a normal function, invoked when the module is already running. Unlike `import` declarations, `require` calls *can* appear inside functions, and the name of the dependency can be any expression that evaluates to a string, whereas `import` only allows plain quoted strings.

The transition of the JavaScript community from CommonJS style to ES modules has been a slow and somewhat rough one. Fortunately we are now at

a point where most of the popular packages on NPM provide their code as ES modules, and Node.js allows ES modules to import from CommonJS modules. While CommonJS code is still something you will run across, there is no real reason to write new programs in this style anymore.

Building and bundling

Many JavaScript packages aren't technically written in JavaScript. Language extensions such as TypeScript, the type checking dialect mentioned in Chapter 8, are widely used. People also often start using planned new language features long before they have been added to the platforms that actually run JavaScript. To make this possible, they *compile* their code, translating it from their chosen JavaScript dialect to plain old JavaScript—or even to a past version of JavaScript—so that browsers can run it.

Including a modular program that consists of 200 different files in a web page produces its own problems. If fetching a single file over the network takes 50 milliseconds, loading the whole program takes 10 seconds, or maybe half that if you can load several files simultaneously. That's a lot of wasted time. Because fetching a single big file tends to be faster than fetching a lot of tiny ones, web programmers have started using tools that combine their programs (which they painstakingly split into modules) into a single big file before they publish it to the web. Such tools are called *bundlers*.

And we can go further. Apart from the number of files, the *size* of the files also determines how fast they can be transferred over the network. Thus, the JavaScript community has invented *minifiers*. These are tools that take a JavaScript program and make it smaller by automatically removing comments and whitespace, renaming bindings, and replacing pieces of code with equivalent code that take up less space.

It is not uncommon for the code that you find in an NPM package or that runs on a web page to have gone through *multiple* stages of transformation—converting from modern JavaScript to historic JavaScript, combining the modules into a single file, and minifying the code. We won't go into the details of these tools in this book since there are many of them, and which one is popular changes regularly. Just be aware that such things exist, and look them up when you need them.

Module design

Structuring programs is one of the subtler aspects of programming. Any nontrivial piece of functionality can be organized in various ways.

Good program design is subjective—there are trade-offs involved, and matters of taste. The best way to learn the value of well-structured design is to read or work on a lot of programs and notice what works and what doesn't. Don't assume that a painful mess is "just the way it is". You can improve the structure of almost everything by putting more thought into it.

One aspect of module design is ease of use. If you are designing something that is intended to be used by multiple people—or even by yourself, in three months when you no longer remember the specifics of what you did—it is helpful if your interface is simple and predictable.

That may mean following existing conventions. A good example is the `ini` package. This module imitates the standard `JSON` object by providing `parse` and `stringify` (to write an INI file) functions, and, like `JSON`, converts between strings and plain objects. The interface is small and familiar, and after you've worked with it once, you're likely to remember how to use it.

Even if there's no standard function or widely used package to imitate, you can keep your modules predictable by using simple data structures and doing a single, focused thing. Many of the INI-file parsing modules on NPM provide a function that directly reads such a file from the hard disk and parses it, for example. This makes it impossible to use such modules in the browser, where we don't have direct file system access, and adds complexity that would have been better addressed by *composing* the module with some file-reading function.

This points to another helpful aspect of module design—the ease with which something can be composed with other code. Focused modules that compute values are applicable in a wider range of programs than bigger modules that perform complicated actions with side effects. An INI file reader that insists on reading the file from disk is useless in a scenario where the file's content comes from some other source.

Relatedly, stateful objects are sometimes useful or even necessary, but if something can be done with a function, use a function. Several of the INI file readers on NPM provide an interface style that requires you to first create an object, then load the file into your object, and finally use specialized methods to get at the results. This type of thing is common in the object-oriented tradition, and it's terrible. Instead of making a single function call and moving on, you have to perform the ritual of moving your object through its various states. And because the data is now wrapped in a specialized object type, all code that interacts with it has to know about that type, creating unnecessary interdependencies.

Often, defining new data structures can't be avoided—only a few basic ones are provided by the language standard, and many types of data have to be more complex than an array or a map. But when an array suffices, use an array.

An example of a slightly more complex data structure is the graph from Chapter 7. There is no single obvious way to represent a graph in JavaScript. In that chapter, we used an object whose properties hold arrays of strings—the other nodes reachable from that node.

There are several different pathfinding packages on NPM, but none of them uses this graph format. They usually allow the graph's edges to have a weight, which is the cost or distance associated with it. That isn't possible in our representation.

For example, there's the `dijkstrajs` package. A well-known approach to pathfinding, quite similar to our `findRoute` function, is called *Dijkstra's algorithm*, after Edsger Dijkstra, who first wrote it down. The `js` suffix is often added to package names to indicate the fact that they are written in JavaScript. This `dijkstrajs` package uses a graph format similar to ours, but instead of arrays, it uses objects whose property values are numbers—the weights of the edges.

If we wanted to use that package, we'd have to make sure that our graph was stored in the format it expects. All edges get the same weight since our simplified model treats each road as having the same cost (one turn).

```
const {find_path} = require("dijkstrajs");

let graph = {};
for (let node of Object.keys(roadGraph)) {
  let edges = graph[node] = {};
  for (let dest of roadGraph[node]) {
    edges[dest] = 1;
  }
}

console.log(find_path(graph, "Post Office", "Cabin"));
// → ["Post Office", "Alice's House", "Cabin"]
```

This can be a barrier to composition—when various packages are using different data structures to describe similar things, combining them is difficult. Therefore, if you want to design for composability, find out what data structures other people are using and, when possible, follow their example.

Designing a fitting module structure for a program can be difficult. In the phase where you are still exploring the problem, trying different things to see what works, you might want to not worry about it too much since keeping everything organized can be a big distraction. Once you have something that feels solid, that's a good time to take a step back and organize it.

Summary

Modules provide structure to bigger programs by separating the code into pieces with clear interfaces and dependencies. The interface is the part of the module that's visible to other modules, and the dependencies are the other modules it makes use of.

Because JavaScript historically did not provide a module system, the CommonJS system was built on top of it. Then at some point it *did* get a built-in system, which now coexists uneasily with the CommonJS system.

A package is a chunk of code that can be distributed on its own. NPM is a repository of JavaScript packages. You can download all kinds of useful (and useless) packages from it.

Exercises

A modular robot

These are the bindings that the project from Chapter 7 creates:

```
roads
buildGraph
roadGraph
VillageState
runRobot
randomPick
randomRobot
mailRoute
routeRobot
findRoute
goalOrientedRobot
```

If you were to write that project as a modular program, what modules would you create? Which module would depend on which other module, and what would their interfaces look like?

Which pieces are likely to be available prewritten on NPM? Would you prefer to use an NPM package or write them yourself?

▶ Display hints...

Roads module

Write an ES module based on the example from Chapter 7 that contains the array of roads and exports the graph data structure representing them as `roadGraph`. It depends on a module `./graph.js` that exports a function `buildGraph`, used to build the graph. This function expects an array of two-element arrays (the start and end points of the roads).

```
// Add dependencies and exports

const roads = [
  "Alice's House-Bob's House",   "Alice's House-Cabin",
  "Alice's House-Post Office",   "Bob's House-Town Hall",
  "Daria's House-Ernie's House", "Daria's House-Town Hall",
```

```
  "Ernie's House-Grete's House", "Grete's House-Farm",
  "Grete's House-Shop",          "Marketplace-Farm",
  "Marketplace-Post Office",     "Marketplace-Shop",
  "Marketplace-Town Hall",       "Shop-Town Hall"
];
```

▶ Display hints...

Circular dependencies

A circular dependency is a situation where module A depends on B, and B also, directly or indirectly, depends on A. Many module systems simply forbid this because whichever order you choose for loading such modules, you cannot make sure that each module's dependencies have been loaded before it runs.

CommonJS modules allow a limited form of cyclic dependencies. As long as the modules don't access each other's interface until after they finish loading, cyclic dependencies are okay.

The `require` function given earlier in this chapter supports this type of dependency cycle. Can you see how it handles cycles?

▶ Display hints...

◂ ● ▸ ?