

Chapter 10. Namespaces.Modules

When you write a program, you can express encapsulation at several levels. At the lowest level, functions encapsulate behaviors, and data structures like objects and lists encapsulate data. You might then group functions and data into classes, or keep them separate as namespaced utilities with a separate database or store for your data. A single class or a set of utilities per file is typical. Going up, you might group a few classes or utilities into a package, which you publish to NPM.

When we talk about modules, it's important to make a distinction between how the compiler (TSC) resolves modules, how your build system (Webpack, Gulp, etc.) resolves modules, and how modules are actually loaded into your application at runtime (`<script />` tags, SystemJS, etc.). In the JavaScript world there is usually a separate program that does each of these jobs, which can make modules hard to reason about. The CommonJS and ES2015 module standards make it easier to interoperate the three programs, and powerful bundlers like Webpack help abstract away the three kinds of resolution happening under the hood.

In this chapter we'll focus on the first of these three kinds of programs: how TypeScript resolves and compiles modules. We'll leave a discussion of how the build system and runtime loaders work with modules to [Chapter 12](#) and talk here about:

- The different ways to namespace and modularize your code
- The different ways to import and export code
- Scaling these approaches as your codebase grows
- Module mode versus script mode
- What declaration merging is, and what you can do with it

But first, a bit of background.

A Brief History of JavaScript Modules

Because TypeScript compiles to and interoperates with JavaScript, it has to support the various module standards that JavaScript programmers use.

In the beginning (in 1995), JavaScript didn't support any sort of module system. Without modules, everything was declared in a global namespace, which made it really hard to build and scale applications. You could quickly run out of variable names, and run into collisions between variable names; and without exposing explicit APIs for each module, it's hard to know which parts of a module you're supposed to use, and which parts are private implementation details.

To help solve these problems, people simulated modules with either objects or *Immediately Invoked Function Expressions* (IIFEs), which they assigned to the global `window`, making them available to other modules in their application (and in other applications hosted on the same web page). It looked something like this:

```
window.emailListModule = {
  renderList() {}
  // ...
}

window.emailComposerModule = {
  renderComposer() {}
  // ...
}

window.appModule = {
  renderApp() {
    window.emailListModule.renderList()
    window.emailComposerModule.renderComposer()
  }
}
```

Because loading and running JavaScript blocks the browser's UI, as a web application grows and includes more and more lines of code, the user's browser gets slower and slower. For this reason, clever programmers started dynamically loading JavaScript after the page loaded, rather than loading it all in one shot. Nearly 10 years after JavaScript was first released, Dojo (Alex

Russell, 2004), YUI (Thomas Sha, 2005), and LABjs (Kyle Simpson, 2009) shipped module loaders—ways to lazily (and often asynchronously) load JavaScript code after the initial page load has happened. Lazy and asynchronous module loading meant three things:

1. Modules needed to be well encapsulated. Otherwise, a page might be broken while dependencies are streaming in.
2. Dependencies between modules needed to be explicit. Otherwise, we don't know which modules need to be loaded and in what order.
3. Every module needed a unique identifier within the app. Otherwise, there's no reliable way to specify what modules need to be loaded.

Loading a module with LABjs looked like this:

```
$LAB
  .script('/emailBaseModule.js').wait()
  .script('/emailListModule.js')
  .script('/emailComposerModule.js')
```

Around the same time, NodeJS (Ryan Dahl, 2009) was being developed, and its creators took a lesson from JavaScript's growing pains and from other languages and decided to build a module system right into the platform. Like any good module system, it needed to satisfy the same three criteria as LABjs and YUI's loaders. NodeJS did that with the CommonJS module standard, which looked like this:

```
// emailBaseModule.js
var emailList = require('emailListModule')
var emailComposer = require('emailComposerModule')

module.exports.renderBase = function() {
  // ...
}
```

In the meantime, on the web the AMD module standard (James Burke, 2008)—pushed by Dojo and RequireJS—was taking off. It supported an equivalent set of functionality, and came with its own build system for bundling up JavaScript code:

```

define('emailBaseModule',
  ['require', 'exports', 'emailListModule', 'emailComposerModule'],
  function(require, exports, emailListModule, emailComposerModule) {
    exports.renderBase = function() {
      // ...
    }
  }
)

```

A few years after that, Browserify came out (James Halliday, 2011), giving frontend engineers the ability to use CommonJS on the frontend, too. CommonJS became the de facto standard for module bundling and import/export syntax.

There were a few problems with the CommonJS way of doing things. Among them, `require` calls are necessarily synchronous, and the CommonJS module resolution algorithm is not ideal for use on the web. On top of that, code that uses it isn't statically analyzable in some cases (as a TypeScript programmer, this should perk your ears up), because `module.exports` can appear anywhere (even in dead code branches that are never actually reached) and `require` calls can appear anywhere and contain arbitrary strings and expressions, making it impossible to statically link a JavaScript program, and verify that all referenced files really exist and export what they say they export.

Against this backdrop, ES2015—the sixth edition of the ECMAScript language—introduced a new standard for imports and exports that had a clean syntax and was statically analyzable. It looks like this:

```

// emailBaseModule.js
import emailList from 'emailListModule'
import emailComposer from 'emailComposerModule'

export function renderBase() {
  // ...
}

```

This is the standard we use in JavaScript and TypeScript code today. However, at the time of writing the standard isn't yet natively supported in every JavaScript runtime, so we have to compile it down to a format that is

supported (CommonJS for NodeJS environments, globals or a module-loadable format for browser environments).

TypeScript gives us a few ways to consume and export code in a module: with global declarations, with standard ES2015 `import` s and `export` s, and with backward-compatible `import` s from CommonJS modules. On top of that, TSC's build system lets us compile modules for a variety of environments: globals, ES2015, CommonJS, AMD, SystemJS, or UMD (a mix of CommonJS, AMD, and globals—whichever happens to be available in the consumer's environment).

import, export

Unless you're being chased by wolves, you should use ES2015 `import` s and `export` s in your TypeScript code, rather than using CommonJS, global, or namespaced modules. They look like this—the same as plain old JavaScript:

```
// a.ts
export function foo() {}
export function bar() {}

// b.ts
import {foo, bar} from './a'
foo()
export let result = bar()
```

The ES2015 module standard supports default exports:

```
// c.ts
export default function meow(loudness: number) {}

// d.ts
import meow from './c' // Note the lack of {curlies}
meow(11)
```



It also supports importing everything from a module using a wildcard import (`*`):

```
// e.ts
import * as a from './a'
a.foo()
a.bar()
```

And reexporting some (or all) exports from a module:

```
// f.ts
export * from './a'
export {result} from './b'
export meow from './c'
```

Because we're writing TypeScript, not JavaScript, we can of course export types and interfaces as well as values. And because types and values live in separate namespaces, it's perfectly fine to export two things—one at the value level and one at the type level—that share the same name. Like for any other code, TypeScript will infer whether you meant the type or the value when you actually use it:

```
// g.ts
export let X = 3
export type X = {y: string}

// h.ts
import {X} from './g'

let a = X + 1 // X refers to the value X
let b: X = {y: 'z'} // X refers to the type X
```

Module paths are filenames on the filesystem. This couples modules with the way they're laid out in the filesystem, but is an important feature for module loaders that need to be aware of that layout so they can resolve module names to files.

Dynamic Imports

As your application gets bigger, its time to initial render will get worse and worse. This is especially a problem for frontend applications where the network can be a bottleneck, but it also applies to backend applications that take more time to start up as you import more code at the top level—code that

needs to be loaded from the filesystem, parsed, compiled, and evaluated, all while blocking other code from running.

On the frontend, one way to deal with this problem (besides writing less code!) is with *code splitting*: chunking your code up into a bunch of generated JavaScript files, instead of shipping everything in a single large file. With splitting you get the benefit of loading multiple chunks in parallel, which eases the toll of large network requests (see [Figure 10-1](#)).

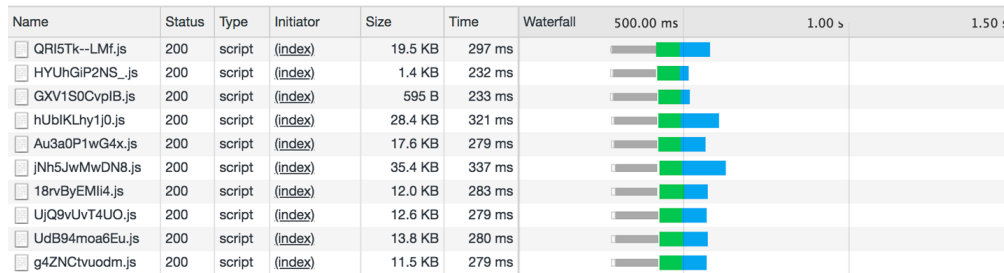


Figure 10-1. Network waterfall for JavaScript loaded from facebook.com

A further optimization is to lazy-load chunks of code when they’re actually needed. Really large frontend applications—like those at Facebook and Google—use this type of optimization as a matter of course. Without it, clients might be loading gigabytes of JavaScript code on the initial page load, which could take minutes or hours (not to mention that people would probably stop using those services once they received their mobile bills).

Lazy loading is also useful for other reasons. For example, the popular [Moment.js](#) date manipulation library comes with packages to support every date format used around the world, split up by locale. Each packages weighs in at around 3 KB. Loading all of these locales for each user might be an unacceptable performance and bandwidth hit; instead, you might want to detect the user’s locale, then load just the relevant date package.

LABjs and its siblings introduced the concept of lazy-loading code when you actually need it, and the concept was formalized in *dynamic imports*. It looks like this:

```
let locale = await import('locale_us-en')
```

You can use `import` either as a statement to statically pull in code (as we’ve used it up to this point), or as a function that returns a `Promise` for your module (as we did in this example).

While you can pass an arbitrary expression that evaluates to a string to `import`, you lose type safety when you do. To safely use dynamic imports, be sure to either:

1. Pass a string literal directly to `import`, without assigning the string to a variable first.
2. Pass an expression to `import` and manually annotate the module's signature.

If using the second option, a common pattern is to statically import the module, but use it only in a type position, so that TypeScript compiles away the static import (to learn more, see [“The types Directive”](#)). For example:

```
import {locale} from './locales/locale-us'

async function main() {
  let userLocale = await getUserLocale()
  let path = './locales/locale-${userLocale}'
  let localeUS: typeof locale = await import(path)
}
```

We imported `locale` from `./locales/locale-us`, but we only used it for its type, which we retrieved with `typeof locale`. We needed to do that because TypeScript couldn't statically look up the type of `import(path)`, because `path` is a computed variable and not a static string. Because we never used `locale` as a value, and instead just scavenged it for its type, TypeScript compiled away the static import (in this example, TypeScript doesn't generate any top-level exports at all), leaving us with both excellent type safety and a dynamically computed import.

TSC SETTING: MODULE

TypeScript supports dynamic imports in `esnext` module mode only. To use dynamic imports, set `{"module": "esnext"}` in your `tsconfig.json`'s `compilerOptions`. Jump ahead to [“Running TypeScript on the Server”](#) and [“Running TypeScript in the Browser”](#) to learn more.

Using CommonJS and AMD Code

When consuming a JavaScript module that uses the CommonJS or AMD standard, you can simply import names from it, just like for ES2015 modules:

```
import {something} from './a/legacy/commonjs/module'
```



By default, CommonJS default exports don't interoperate with ES2015 default imports; to use a default export, you have to use a wildcard import:

```
import * as fs from 'fs'
fs.readFile('some/file.txt')
```

To interoperate more smoothly, set `{"esModuleInterop": true}` in your `tsconfig.json`'s `compilerOptions`. Now, you can leave out the wildcard:

```
import fs from 'fs'
fs.readFile('some/file.txt')
```

NOTE

As I mentioned at the top of the chapter, even though this code compiles, that doesn't mean it'll work at runtime. Whichever module standard you use— `import / export`, CommonJS, AMD, UMD, or browser globals—your module bundler and module loader have to be aware of that format so they can package up and split your code correctly at compile time, and load your code correctly at runtime. Head over to [Chapter 12](#) to learn more.

Module Mode Versus Script Mode

TypeScript parses each of your TypeScript files in one of two modes: *module mode* or *script mode*. It decides which mode to use based on a single heuristic: does your file have any `import` s or `export` s? If so, it uses module mode; otherwise, it uses script mode.

Module mode is what we've used up to this point, and what you'll use most of the time. In module mode, you use `import` and `import()` to require code

from other files, and `export` to make code available to other files. If you use any third-party UMD modules (as a reminder, UMD modules try to use CommonJS, RequireJS, or browser globals, whichever the environment supports), you have to `import` them first, and can't use their global exports directly.

In script mode, any top-level variables you declare will be available to other files in your project without an explicit import, and you can safely consume global exports from third-party UMD modules without explicitly importing them first. A couple of use cases for script mode are:

- To quickly prototype browser code that you plan to compile to no module system at all (`{ "module": "none" }` in your *tsconfig.json*) and include as raw `<script />` tags in your HTML file.
- To create type declarations (see [“Type Declarations”](#))

You'll almost always want to stick to module mode, which TypeScript will choose for you automatically as you write real-world code that `import`s other code and `export`s things for other files to use.

Namespaces

TypeScript gives us another way to encapsulate code: the `namespace` keyword. Namespaces will feel familiar to a lot of Java, C#, C++, PHP, and Python programmers.

TIP

If you're coming from a language with namespaces, note that although namespaces are supported by TypeScript, they're not the preferred way to encapsulate code; if you're not sure whether to use namespaces or modules, choose modules.

Namespaces abstract away the nitty-gritty details of how files are laid out in the filesystem; you don't have to know that your `.mine` function lives in the `schemes/scams/bitcoin/apps` folder, and instead you can access it with a short, convenient namespace like `Schemes.Scams.Bitcoin.Apps.mine`.¹

Say we have two files—a module to make HTTP GET requests, and a consumer that uses that module to make requests:

```
// Get.ts
namespace Network {
  export function get<T>(url: string): Promise<T> {
    // ...
  }
}

// App.ts
namespace App {
  Network.get<GitRepo>('https://api.github.com/repos/Mi
}
```

A namespace must have a name (like `Network`), and it can export functions, variables, types, interfaces, or other namespaces. Any code in a `namespace` block that's not explicitly exported is private to the block. Because namespaces can export namespaces, you can easily model nested namespaces. Let's say our `Network` module is getting big, and we want to split it up into a few submodules. We can use namespaces to do that:

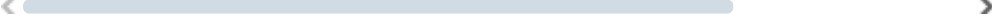
```
namespace Network {
  export namespace HTTP {
    export function get <T>(url: string): Promise
      // ...
  }
  export namespace TCP {
    listenOn(port: number): Connection {
      //...
    }
    // ...
  }
  export namespace UDP {
    // ...
  }
  export namespace IP {
    // ...
  }
}
```

Now, all of our network-related utilities are in subnamespaces under `Network`. For example, we can now call `Network.HTTP.get` and `Network.TCP.listenOn` from any file. Like interfaces, namespaces can be augmented, making it convenient to split them across files. TypeScript will recursively merge identically named namespaces for us:

```
// HTTP.ts
namespace Network {
  export namespace HTTP {
    export function get<T>(url: string): Promise<T> {
      // ...
    }
  }
}

// UDP.ts
namespace Network {
  export namespace UDP {
    export function send(url: string, packets: Buffer):
      // ...
  }
}

// MyApp.ts
Network.HTTP.get<Dog[]>('http://url.com/dogs')
Network.UDP.send('http://url.com/cats', new Buffer(123))
```



If you end up with long namespace hierarchies, you can use *aliases* to shorten them for convenience. Note that despite the similar syntax, destructuring (like you do when importing ES2015 modules) is not supported for aliases:

```
// A.ts
namespace A {
  export namespace B {
    export namespace C {
      export let d = 3
    }
  }
}

// MyApp.ts
```

```
import d = A.B.C.d
```

```
let e = d * 3
```

Collisions

Collisions between identically named exports are not allowed:

```
// HTTP.ts
namespace Network {
  export function request<T>(url: string): T {
    // ...
  }
}

// HTTP2.ts
namespace Network {
  // Error TS2393: Duplicate function implementation.
  export function request<T>(url: string): T {
    // ...
  }
}
```



The exception to the no-collisions rule is overloaded ambient function declarations, which you can use to refine function types:

```
// HTTP.ts
namespace Network {
  export function request<T>(url: string): T
}

// HTTP2.ts
namespace Network {
  export function request<T>(url: string, priority: number): T
}

// HTTPS.ts
namespace Network {
  export function request<T>(url: string, algo: 'SHA1')
}
```



Compiled Output

Unlike imports and exports, namespaces don't respect your *tsconfig.json*'s `module` setting, and always compile to global variables. Let's peek behind the veil to see what the generated output looks like. Say we have the following module:

```
// Flowers.ts
namespace Flowers {
  export function give(count: number) {
    return count + ' flowers'
  }
}
```

Running it through TSC, the generated JavaScript output looks like this:

```
let Flowers
(function (Flowers) {
  ❶

  function give(count) {
    return count + ' flowers'
  }
  Flowers.give = give
  ❷

})(Flowers || (Flowers = {}))
  ❸
```

`Flowers` is declared within an IIFE—❶ a function that calls itself immediately—to create a closure and prevent variables that weren't explicitly exported from leaking out of the `Flowers` module.

TypeScript assigns the `give` ❷ function that we exported to the `Flowers` namespace.

If the `Flowers` namespace is already globally defined, then ❸ TypeScript augments it (`Flowers`); otherwise, TypeScript creates and augments that newly created namespace (`Flowers = {}`).

Prefer regular modules (the `import` and `export` kind) over namespaces as a way to more closely stick to JavaScript standards and make your dependencies more explicit.

Explicit dependencies have lots of benefits for readability, enforcing module isolation (because namespaces are automatically merged, but modules are not), and static analysis, which matters for big frontend projects where stripping out dead code and splitting your compiled code into multiple files is crucial for performance.

When running TypeScript programs in a NodeJS environment, modules are also the clear choice because of NodeJS's built-in support for CommonJS. In browser environments, some programmers prefer namespaces for simplicity, but for medium- to large-sized projects, try to stick to modules over namespaces.

Declaration Merging

So far we've touched on three types of merging that TypeScript does for us:

- Merging values and types, so that the same name can refer to either a value or a type depending how we use it (see [“Companion Object Pattern”](#))
- Merging multiple namespaces into one
- Merging multiple interfaces into one (see [“Declaration Merging”](#))

As you might have intuited, these are three special cases of a much more general TypeScript behavior. TypeScript has a rich set of behavior for merging different kinds of names, unlocking all sorts of patterns that can otherwise be difficult to express (see [Table 10-1](#)).

Table 10-1. Can the declaration be merged?

						To
		Value	Class	Enum	Function	Types alias
From	Value	No	No	No	No	Yes
	Class	—	No	No	No	No
	Enum	—	—	Yes	No	No
	Function	—	—	—	No	Yes
	Type alias	—	—	—	—	No
	Interface	—	—	—	—	—
	Namespace	—	—	—	—	—
	Module	—	—	—	—	—

This means that if, for example, you declare a value and a type alias in the same scope, TypeScript will allow it, and infer which one you meant—the type or the value—from whether you use the name in a value or a type position. This is what lets us implement the pattern described in [“Companion Object Pattern”](#). It also means that you can use an interface and a namespace to implement companion objects—you’re not limited to just a value and a type alias. Or you can take advantage of module merging to augment a third-party module declaration (more on this in [“Extending a Module”](#)). Or you can add static methods to an enum by merging that enum with a namespace (try it!).

Eagle-eyed readers may notice the `moduleResolution` flag available in their `tsconfig.json`. This flag controls the algorithm TypeScript uses to resolve module names in your application. The flag supports two modes:

- **node** : Always use this mode. It resolves modules using the same algorithm that NodeJS uses. Modules prefixed with a `.`, `/`, or `~` (like `./my/file`) are resolved from the local filesystem, either relative to the current file, or using an absolute path (relative to your `/` directory, or whatever your `tsconfig.json`'s `baseUrl` is set to), depending on the prefix you use. TypeScript loads module paths that don't have a prefix from your `node_modules` folder, the same as NodeJS. TypeScript builds on NodeJS's resolution strategy in two ways:
 1. In addition to the `main` field in a package's `package.json` that NodeJS looks at to find the default importable file in a directory, TypeScript also looks at the TypeScript-specific `types` property (more on that in [“Type Lookup for JavaScript”](#)).
 2. When importing a file with an unspecified extension, TypeScript first looks for a file with that name and a `.ts` extension, followed by `.tsx`, `.d.ts`, and finally `.js`.
 - **classic** : You should never use this mode. In this mode, relative paths are resolved like in `node` mode, but for unprefixed names, TypeScript will look for a file with the given name in the current folder, then walk up the directory tree a folder at a time until it finds a matching file. This is really surprising behavior for anyone coming from the NodeJS or JavaScript world, and does not interoperate well with other build tools.
-

Summary

In this chapter we covered TypeScript's module system, starting with a brief history of JavaScript module systems, ES2015 modules and safely lazy-loading code with dynamic imports, interoperating with CommonJS and AMD modules, and module mode versus script mode. We then covered namespaces, namespace merging, and how TypeScript's declaration merging works.

As you develop applications in TypeScript, try hard to stick to ES2015 modules. TypeScript doesn't care which module system you use, but it will make it easier to integrate with build tooling (see [Chapter 12](#) to learn more).

Exercise

1. Play around with declaration merging, to:

1. Reimplement companion objects (from [“Companion Object Pattern”](#)) using namespaces and interfaces, instead of values and types.
2. Add static methods to an enum.

1 I really hope this joke ages well, and I don't end up regretting not investing in Bitcoin.