Namespaces and Modules

> **A note about terminology:** It's important to note that in TypeScript 1.5, the nomenclature has changed. "Internal modules" are now "namespaces". "External modules" are now simply "modules", as to align with ECMAScript 2015 (http://www.ecma-international.org/ecma-262/6.0/)'s terminology, (namely that `module X {` is equivalent to the now-preferred `namespace X {`).

Introduction

This post outlines the various ways to organize your code using namespaces and modules in TypeScript. We'll also go over some advanced topics of how to use namespaces and modules, and address some common pitfalls when using them in TypeScript.

See the Modules (./modules.html) documentation for more information about modules. See the Namespaces (./namespaces.html) documentation for more information about namespaces.

Using Namespaces

Namespaces are simply named JavaScript objects in the global namespace. This makes namespaces a very simple construct to use. They can span multiple files, and can be concatenated using `--outFile` . Namespaces can be a good way to structure your code in a Web Application, with all dependencies included as `<script>` tags in your HTML page.

Just like all global namespace pollution, it can be hard to identify component dependencies, especially in a large application.

Using Modules

Just like namespaces, modules can contain both code and declarations. The main difference is that modules *declare* their dependencies.

Modules also have a dependency on a module loader (such as CommonJs/Require.js). For a small JS application this might not be optimal, but for larger applications, the cost comes with long term modularity and maintainability benefits. Modules provide for better code reuse, stronger isolation and better tooling support for bundling.

It is also worth noting that, for Node.js applications, modules are the default and the recommended approach to structure your code.

Starting with ECMAScript 2015, modules are native part of the language, and should be supported by all compliant engine implementations. Thus, for new projects modules would be the recommended code organization mechanism.

Pitfalls of Namespaces and Modules

In this section we'll describe various common pitfalls in using namespaces and modules, and how to avoid them.

`/// <reference>` -ing a module

A common mistake is to try to use the `/// <reference ... />` syntax to refer to a module file, rather than using an `import` statement. To understand the distinction, we first need to understand how the compiler can locate the type information for a module based on the path of an `import` (e.g. the `...` in `import x from "...";`, `import x = require("...");`, etc.) path.

The compiler will try to find a `.ts`, `.tsx`, and then a `.d.ts` with the appropriate path. If a specific file could not be found, then the compiler will look for an *ambient module declaration*. Recall that these need to be declared in a `.d.ts` file.

- `myModules.d.ts`

```
// In a .d.ts file or .ts file that is not a module:
declare module "SomeModule" {
    export function fn(): string;
}
```

- `myOtherModule.ts`

```
/// <reference path="myModules.d.ts" />
import * as m from "SomeModule";
```

The reference tag here allows us to locate the declaration file that contains the declaration for the ambient module. This is how the `node.d.ts` file that several of the TypeScript samples use is consumed.

Needless Namespacing

If you're converting a program from namespaces to modules, it can be easy to end up with a file that looks like this:

- `shapes.ts`

```
export namespace Shapes {
    export class Triangle { /* ... */ }
    export class Square { /* ... */ }
}
```

The top-level module here `Shapes` wraps up `Triangle` and `Square` for no reason. This is confusing and annoying for consumers of your module:

- `shapeConsumer.ts`

```
import * as shapes from "./shapes";
let t = new shapes.Shapes.Triangle(); // shapes.Shapes?
```

A key feature of modules in TypeScript is that two different modules will never contribute names to the same scope. Because the consumer of a module decides what name to assign it, there's no need to proactively wrap up the exported symbols in a namespace.

To reiterate why you shouldn't try to namespace your module contents, the general idea of namespacing is to provide logical grouping of constructs and to prevent name collisions. Because the module file itself is already a logical grouping, and its top-level name is defined by the code that imports it, it's unnecessary to use an additional module layer for exported objects.

Here's a revised example:

- `shapes.ts`

```
export class Triangle { /* ... */ }
export class Square { /* ... */ }
```

- `shapeConsumer.ts`

```
import * as shapes from "./shapes";
let t = new shapes.Triangle();
```

Trade-offs of Modules

Just as there is a one-to-one correspondence between JS files and modules, TypeScript has a one-to-one correspondence between module source files and their emitted JS files. One effect of this is that it's not possible to concatenate multiple module source files depending on the module system you target. For instance, you can't use the `outFile` option while targeting `commonjs` or `umd` , but with TypeScript 1.8 and later, it's possible (./release-notes/typescript-1-8.html#concatenate-amd-and-system-modules-with---outfile) to use `outFile` when targeting `amd` or `system` .