Skip to main content

TypeScript 3.2 is now available. Download (/#download-links) our latest version today!

Documentation ▾

Namespaces

> **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 (previously "internal modules") in TypeScript. As we alluded in our note about terminology, "internal modules" are now referred to as "namespaces". Additionally, anywhere the `module` keyword was used when declaring an internal module, the `namespace` keyword can and should be used instead. This avoids confusing new users by overloading them with similarly named terms.

First steps

Let's start with the program we'll be using as our example throughout this page. We've written a small set of simplistic string validators, as you might write to check a user's input on a form in a webpage or check the format of an externally-provided data file.

Validators in a single file

```
interface StringValidator {
    isAcceptable(s: string): boolean;
}

let lettersRegexp = /^[A-Za-z]+$/;
let numberRegexp = /^[0-9]+$/;

class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}

class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: StringValidator; } = {};
validators["ZIP code"] = new ZipCodeValidator();
validators["Letters only"] = new LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        let isMatch = validators[name].isAcceptable(s);
        console.log(`'${ s }' ${ isMatch ? "matches" : "does not match" } '${ name }'.`);
    }
}
```

Namespacing

As we add more validators, we're going to want to have some kind of organization scheme so that we can keep track of our types and not worry about name collisions with other objects. Instead of putting lots of different names into the global namespace, let's wrap up our objects into a namespace.

In this example, we'll move all validator-related entities into a namespace called `Validation`. Because we want the interfaces and classes here to be visible outside the namespace, we preface them with `export`. Conversely, the variables `lettersRegexp` and `numberRegexp` are implementation details, so they are left unexported and will not be visible to code outside the namespace. In the test code at the bottom of the file, we now need to qualify the names of the types when used outside the namespace, e.g. `Validation.LettersOnlyValidator`.

Namespaced Validators

```
namespace Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }

    const lettersRegexp = /^[A-Za-z]+$/;
    const numberRegexp = /^[0-9]+$/;

    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }

    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: Validation.StringValidator; } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        console.log(`"${ s }" - ${ validators[name].isAcceptable(s) ? "matches" : "does not match"
 } ${ name }`);
    }
}
```

Splitting Across Files

As our application grows, we'll want to split the code across multiple files to make it easier to maintain.

Multi-file namespaces

Here, we'll split our `Validation` namespace across many files. Even though the files are separate, they can each contribute to the same namespace and can be consumed as if they were all defined in one place. Because there are dependencies between files, we'll add reference tags to tell the compiler about the relationships between the files. Our test code is otherwise unchanged.

Validation.ts

```
namespace Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }
}
```

LettersOnlyValidator.ts

```
/// <reference path="Validation.ts" />
namespace Validation {
    const lettersRegexp = /^[A-Za-z]+$/;
    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }
}
```

ZipCodeValidator.ts

```
/// <reference path="Validation.ts" />
namespace Validation {
    const numberRegexp = /^[0-9]+$/;
    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}
```

Test.ts

```
/// <reference path="Validation.ts" />
/// <reference path="LettersOnlyValidator.ts" />
/// <reference path="ZipCodeValidator.ts" />

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: Validation.StringValidator; } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        console.log(`"${ s }" - ${ validators[name].isAcceptable(s) ? "matches" : "does not match"
 } ${ name }`);
    }
}
```

Once there are multiple files involved, we'll need to make sure all of the compiled code gets loaded. There are two ways of doing this.

First, we can use concatenated output using the `--outFile` flag to compile all of the input files into a single JavaScript output file:

```
tsc --outFile sample.js Test.ts
```

The compiler will automatically order the output file based on the reference tags present in the files. You can also specify each file individually:

```
tsc --outFile sample.js Validation.ts LettersOnlyValidator.ts ZipCodeValidator.ts Test.ts
```

Alternatively, we can use per-file compilation (the default) to emit one JavaScript file for each input file. If multiple JS files get produced, we'll need to use `<script>` tags on our webpage to load each emitted file in the appropriate order, for example:

MyTestPage.html (excerpt)

```
    <script src="Validation.js" type="text/javascript" />
    <script src="LettersOnlyValidator.js" type="text/javascript" />
    <script src="ZipCodeValidator.js" type="text/javascript" />
    <script src="Test.js" type="text/javascript" />
```

Aliases

Another way that you can simplify working with of namespaces is to use `import q = x.y.z` to create shorter names for commonly-used objects. Not to be confused with the `import x = require("name")` syntax used to load modules, this syntax simply creates an alias for the specified symbol. You can use these sorts of imports (commonly referred to as aliases) for any kind of identifier, including objects created from module imports.

```
namespace Shapes {
    export namespace Polygons {
        export class Triangle { }
        export class Square { }
    }
}

import polygons = Shapes.Polygons;
let sq = new polygons.Square(); // Same as 'new Shapes.Polygons.Square()'
```

Notice that we don't use the `require` keyword; instead we assign directly from the qualified name of the symbol we're importing. This is similar to using `var`, but also works on the type and namespace meanings of the imported symbol. Importantly, for values, `import` is a distinct reference from the original symbol, so changes to an aliased `var` will not be reflected in the original variable.

Working with Other JavaScript Libraries

To describe the shape of libraries not written in TypeScript, we need to declare the API that the library exposes. Because most JavaScript libraries expose only a few top-level objects, namespaces are a good way to represent them.

We call declarations that don't define an implementation "ambient". Typically these are defined in `.d.ts` files. If you're familiar with C/C++, you can think of these as `.h` files. Let's look at a few examples.

Ambient Namespaces

The popular library D3 defines its functionality in a global object called `d3`. Because this library is loaded through a `<script>` tag (instead of a module loader), its declaration uses namespaces to define its shape. For the TypeScript compiler to see this shape, we use an ambient namespace declaration. For example, we could begin writing it as follows:

D3.d.ts (simplified excerpt)

```
declare namespace D3 {
    export interface Selectors {
        select: {
            (selector: string): Selection;
            (element: EventTarget): Selection;
        };
    }

    export interface Event {
        x: number;
        y: number;
    }

    export interface Base extends Selectors {
        event: Event;
    }
}

declare var d3: D3.Base;
```