

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

Documentation ▼

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

Starting with ECMAScript 2015, JavaScript has a concept of modules. TypeScript shares this concept.

Modules are executed within their own scope, not in the global scope; this means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported using one of the `export` forms. Conversely, to consume a variable, function, class, interface, etc. exported from a different module, it has to be imported using one of the `import` forms.

Modules are declarative; the relationships between modules are specified in terms of imports and exports at the file level.

Modules import one another using a module loader. At runtime the module loader is responsible for locating and executing all dependencies of a module before executing it. Well-known modules loaders used in JavaScript are the CommonJS (<https://en.wikipedia.org/wiki/CommonJS>) module loader for Node.js and `require.js` (<http://requirejs.org/>) for Web applications.

In TypeScript, just as in ECMAScript 2015, any file containing a top-level `import` or `export` is considered a module. Conversely, a file without any top-level `import` or `export` declarations is treated as a script whose contents are available in the global scope (and therefore to modules as well).

Export

Exporting a declaration

Any declaration (such as a variable, function, class, type alias, or interface) can be exported by adding the `export` keyword.

Validation.ts

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

ZipCodeValidator.ts

```
export const numberRegex = /^[0-9]+$/;

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

Export statements

Export statements are handy when exports need to be renamed for consumers, so the above example can be written as:

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

Re-exports

Often modules extend other modules, and partially expose some of their features. A re-export does not import it locally, or introduce a local variable.

ParseIntBasedZipCodeValidator.ts

```
export class ParseIntBasedZipCodeValidator {
  isAcceptable(s: string) {
    return s.length === 5 && parseInt(s).toString() === s;
  }
}

// Export original validator but rename it
export { ZipCodeValidator as RegExpBasedZipCodeValidator } from "./ZipCodeValidator";
```

Optionally, a module can wrap one or more modules and combine all their exports using `export * from "module"` syntax.

AllValidators.ts

```
export * from "./StringValidator"; // exports interface 'StringValidator'
export * from "./LettersOnlyValidator"; // exports class 'LettersOnlyValidator'
export * from "./ZipCodeValidator"; // exports class 'ZipCodeValidator'
```

Import

Importing is just about as easy as exporting from a module. Importing an exported declaration is done through using one of the `import` forms below:

Import a single export from a module

```
import { ZipCodeValidator } from "./ZipCodeValidator";

let myValidator = new ZipCodeValidator();
```

imports can also be renamed

```
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
let myValidator = new ZCV();
```

Import the entire module into a single variable, and use it to access the module exports

```
import * as validator from "./ZipCodeValidator";
let myValidator = new validator.ZipCodeValidator();
```

Import a module for side-effects only

Though not recommended practice, some modules set up some global state that can be used by other modules. These modules may not have any exports, or the consumer is not interested in any of their exports. To import these modules, use:

```
import "./my-module.js";
```

Default exports

Each module can optionally export a `default` export. Default exports are marked with the keyword `default` ; and there can only be one `default` export per module. `default` exports are imported using a different import form.

`default` exports are really handy. For instance, a library like JQuery might have a default export of `jQuery` or `$` , which we'd probably also import under the name `$` or `jQuery` .

JQuery.d.ts

```
declare let $: JQuery;
export default $;
```

App.ts

```
import $ from "jQuery";

$("button.continue").html( "Next Step..." );
```

Classes and function declarations can be authored directly as default exports. Default export class and function declaration names are optional.

ZipCodeValidator.ts

```
export default class ZipCodeValidator {
    static numberRegexp = /^[0-9]+$/;
    isAcceptable(s: string) {
        return s.length === 5 && ZipCodeValidator.numberRegexp.test(s);
    }
}
```

Test.ts

```
import validator from "./ZipCodeValidator";

let myValidator = new validator();
```

or

StaticZipCodeValidator.ts

```
const numberRegex = /^[0-9]+$/;

export default function (s: string) {
    return s.length === 5 && numberRegex.test(s);
}
```

Test.ts

```
import validate from "./StaticZipCodeValidator";

let strings = ["Hello", "98052", "101"];

// Use function validate
strings.forEach(s => {
    console.log(`"${s}" ${validate(s) ? " matches" : " does not match"}`);
});
```

default exports can also be just values:

OneTwoThree.ts

```
export default "123";
```

Log.ts

```
import num from "./OneTwoThree";

console.log(num); // "123"
```

export = and import = require()

Both CommonJS and AMD generally have the concept of an `exports` object which contains all exports from a module.

They also support replacing the `exports` object with a custom single object. Default exports are meant to act as a replacement for this behavior; however, the two are incompatible. TypeScript supports `export =` to model the traditional CommonJS and AMD workflow.

The `export =` syntax specifies a single object that is exported from the module. This can be a class, interface, namespace, function, or enum.

When exporting a module using `export =`, TypeScript-specific `import module = require("module")` must be used to import the module.

ZipCodeValidator.ts

```
let numberRegex = /^[0-9]+$/;
class ZipCodeValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegex.test(s);
    }
}
export = ZipCodeValidator;
```

Test.ts

```
import zip = require("./ZipCodeValidator");

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

// Validators to use
let validator = new zip();

// Show whether each string passed each validator
strings.forEach(s => {
    console.log(`${ s }" - ${ validator.isAcceptable(s) ? "matches" : "does not match" }`);
});
```

Code Generation for Modules

Depending on the module target specified during compilation, the compiler will generate appropriate code for Node.js (CommonJS (<http://wiki.commonjs.org/wiki/CommonJS>)), require.js (AMD (<https://github.com/amdjs/amdjs-api/wiki/AMD>)), UMD (<https://github.com/umdjs/umd>), SystemJS (<https://github.com/systemjs/systemjs>), or ECMAScript 2015 native modules (<http://www.ecma-international.org/ecma-262/6.0/#sec-modules>) (ES6) module-loading systems. For more information on what the `define`, `require` and `register` calls in the generated code do, consult the documentation for each module loader.

This simple example shows how the names used during importing and exporting get translated into the module loading code.

SimpleModule.ts

```
import m = require("mod");
export let t = m.something + 1;
```

AMD / RequireJS SimpleModule.js

```
define(["require", "exports", "./mod"], function (require, exports, mod_1) {
    exports.t = mod_1.something + 1;
});
```

CommonJS / Node SimpleModule.js

```
var mod_1 = require("./mod");
exports.t = mod_1.something + 1;
```

UMD SimpleModule.js

```
(function (factory) {
    if (typeof module === "object" && typeof module.exports === "object") {
        var v = factory(require, exports); if (v !== undefined) module.exports = v;
    }
    else if (typeof define === "function" && define.amd) {
        define(["require", "exports", "./mod"], factory);
    }
})(function (require, exports) {
    var mod_1 = require("./mod");
    exports.t = mod_1.something + 1;
});
```

System SimpleModule.js

```
System.register(["./mod"], function(exports_1) {
    var mod_1;
    var t;
    return {
        setters:[
            function (mod_1_1) {
                mod_1 = mod_1_1;
            },
        ],
        execute: function() {
            exports_1("t", t = mod_1.something + 1);
        }
    }
});
```

Native ECMAScript 2015 modules SimpleModule.js

```
import { something } from "./mod";
export var t = something + 1;
```

Simple Example

Below, we've consolidated the Validator implementations used in previous examples to only export a single named export from each module.

To compile, we must specify a module target on the command line. For Node.js, use `--module commonjs` ; for require.js, use `--module amd` . For example:

```
tsc --module commonjs Test.ts
```

When compiled, each module will become a separate `.js` file. As with reference tags, the compiler will follow `import` statements to compile dependent files.

Validation.ts

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

LettersOnlyValidator.ts

```
import { StringValidator } from "./Validation";

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

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

ZipCodeValidator.ts

```
import { StringValidator } from "./Validation";

const numberRegex = /^[0-9]+$/;

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

Test.ts

```
import { StringValidator } from "./Validation";
import { ZipCodeValidator } from "./ZipCodeValidator";
import { LettersOnlyValidator } from "./LettersOnlyValidator";

// 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
strings.forEach(s => {
    for (let name in validators) {
        console.log(`${s} - ${validators[name].isAcceptable(s) ? "matches" : "does not match"}`);
    }
});
```

Optional Module Loading and Other Advanced Loading Scenarios

In some cases, you may want to only load a module under some conditions. In TypeScript, we can use the pattern shown below to implement this and other advanced loading scenarios to directly invoke the module loaders without losing type safety.

The compiler detects whether each module is used in the emitted JavaScript. If a module identifier is only ever used as part of a type annotations and never as an expression, then no `require` call is emitted for that module. This elision of unused references is a good performance optimization, and also allows for optional loading of those modules.

The core idea of the pattern is that the `import id = require("...")` statement gives us access to the types exposed by the module. The module loader is invoked (through `require`) dynamically, as shown in the `if` blocks below. This leverages the reference-elision optimization so that the module is only loaded when needed. For this pattern to work, it's important that the symbol defined via an `import` is only used in type positions (i.e. never in a position that would be emitted into the JavaScript).

To maintain type safety, we can use the `typeof` keyword. The `typeof` keyword, when used in a type position, produces the type of a value, in this case the type of the module.

Dynamic Module Loading in Node.js

```

declare function require(moduleName: string): any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
    let ZipCodeValidator: typeof Zip = require("./ZipCodeValidator");
    let validator = new ZipCodeValidator();
    if (validator.isAcceptable("...")) { /* ... */ }
}

```

Sample: Dynamic Module Loading in require.js

```

declare function require(moduleNames: string[], onLoad: (...args: any[]) => void): void;

import * as Zip from "./ZipCodeValidator";

if (needZipValidation) {
    require(["./ZipCodeValidator"], (ZipCodeValidator: typeof Zip) => {
        let validator = new ZipCodeValidator.ZipCodeValidator();
        if (validator.isAcceptable("...")) { /* ... */ }
    });
}

```

Sample: Dynamic Module Loading in System.js

```

declare const System: any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
    System.import("./ZipCodeValidator").then((ZipCodeValidator: typeof Zip) => {
        var x = new ZipCodeValidator();
        if (x.isAcceptable("...")) { /* ... */ }
    });
}

```

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.

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 Modules

In Node.js, most tasks are accomplished by loading one or more modules. We could define each module in its own `.d.ts` file with top-level export declarations, but it's more convenient to write them as one larger `.d.ts` file. To do so, we use a construct similar to ambient namespaces, but we use the `module` keyword and the quoted name of the module which will be available to a later import. For example:

node.d.ts (simplified excerpt)


```

declare module "url" {
    export interface Url {
        protocol?: string;
        hostname?: string;
        pathname?: string;
    }

    export function parse(urlStr: string, parseQueryString?, slashesDenoteHost?): Url;
}

declare module "path" {
    export function normalize(p: string): string;
    export function join(...paths: any[]): string;
    export var sep: string;
}

```

Now we can `/// <reference> node.d.ts` and then load the modules using `import url = require("url");` or `import * as URL from "url"`.

```

/// <reference path="node.d.ts"/>
import * as URL from "url";
let myUrl = URL.parse("http://www.typescriptlang.org");

```

Shorthand ambient modules

If you don't want to take the time to write out declarations before using a new module, you can use a shorthand declaration to get started quickly.

declarations.d.ts

```
declare module "hot-new-module";
```

All imports from a shorthand module will have the `any` type.

```
import x, {y} from "hot-new-module";
x(y);
```

Wildcard module declarations

Some module loaders such as SystemJS (<https://github.com/systemjs/systemjs/blob/master/docs/overview.md#plugin-syntax>) and AMD (<https://github.com/amdjs/amdjs-api/blob/master/LoaderPlugins.md>) allow non-JavaScript content to be imported. These typically use a prefix or suffix to indicate the special loading semantics. Wildcard module declarations can be used to cover these cases.

```

declare module "?!text" {
    const content: string;
    export default content;
}
// Some do it the other way around.
declare module "json!*" {
    const value: any;
    export default value;
}

```

Now you can import things that match `"?!text"` or `"json!*"`.

```
import fileContent from "./xyz.txt!text";
import data from "json!http://example.com/data.json";
console.log(data, fileContent);
```

UMD modules

Some libraries are designed to be used in many module loaders, or with no module loading (global variables). These are known as UMD (<https://github.com/umdjs/umd>) modules. These libraries can be accessed through either an import or a global variable. For example:

math-lib.d.ts

```
export function isPrime(x: number): boolean;
export as namespace mathLib;
```

The library can then be used as an import within modules:

```
import { isPrime } from "math-lib";
isPrime(2);
mathLib.isPrime(2); // ERROR: can't use the global definition from inside a module
```

It can also be used as a global variable, but only inside of a script. (A script is a file with no imports or exports.)

```
mathLib.isPrime(2);
```

Guidance for structuring modules

Export as close to top-level as possible

Consumers of your module should have as little friction as possible when using things that you export. Adding too many levels of nesting tends to be cumbersome, so think carefully about how you want to structure things.

Exporting a namespace from your module is an example of adding too many layers of nesting. While namespaces sometime have their uses, they add an extra level of indirection when using modules. This can quickly become a pain point for users, and is usually unnecessary.

Static methods on an exported class have a similar problem - the class itself adds a layer of nesting. Unless it increases expressivity or intent in a clearly useful way, consider simply exporting a helper function.

If you're only exporting a single class or function, use export default

Just as "exporting near the top-level" reduces friction on your module's consumers, so does introducing a default export. If a module's primary purpose is to house one specific export, then you should consider exporting it as a default export. This makes both importing and actually using the import a little easier. For example:

MyClass.ts

```
export default class SomeType {
  constructor() { ... }
}
```

MyFunc.ts

```
export default function getThing() { return "thing"; }
```

Consumer.ts

```
import t from "./MyClass";
import f from "./MyFunc";
let x = new t();
console.log(f());
```

This is optimal for consumers. They can name your type whatever they want (`t` in this case) and don't have to do any excessive dotting to find your objects.

If you're exporting multiple objects, put them all at top-level

MyThings.ts

```
export class SomeType { /* ... */ }
export function someFunc() { /* ... */ }
```

Conversely when importing:

Explicitly list imported names

Consumer.ts

```
import { SomeType, someFunc } from "./MyThings";
let x = new SomeType();
let y = someFunc();
```

Use the namespace import pattern if you're importing a large number of things

MyLargeModule.ts

```
export class Dog { ... }
export class Cat { ... }
export class Tree { ... }
export class Flower { ... }
```

Consumer.ts

```
import * as myLargeModule from "./MyLargeModule.ts";
let x = new myLargeModule.Dog();
```

Re-export to extend

Often you will need to extend functionality on a module. A common JS pattern is to augment the original object with *extensions*, similar to how JQuery extensions work. As we've mentioned before, modules do not *merge* like global namespace objects would. The recommended solution is to *not* mutate the original object, but rather export a new entity that provides the new functionality.

Consider a simple calculator implementation defined in module `Calculator.ts`. The module also exports a helper function to test the calculator functionality by passing a list of input strings and writing the result at the end.

Calculator.ts

```
export class Calculator {
    private current = 0;
    private memory = 0;
    private operator: string;

    protected processDigit(digit: string, currentValue: number) {
        if (digit >= "0" && digit <= "9") {
            return currentValue * 10 + (digit.charCodeAt(0) - "0".charCodeAt(0));
        }
    }

    protected processOperator(operator: string) {
        if (["+ ", "- ", " * ", " / "].indexOf(operator) >= 0) {
            return operator;
        }
    }

    protected evaluateOperator(operator: string, left: number, right: number): number {
        switch (this.operator) {
            case "+": return left + right;
            case "-": return left - right;
            case "*": return left * right;
            case "/": return left / right;
        }
    }

    private evaluate() {
        if (this.operator) {
            this.memory = this.evaluateOperator(this.operator, this.memory, this.current);
        }
        else {
            this.memory = this.current;
        }
        this.current = 0;
    }

    public handleChar(char: string) {
        if (char === "=") {
            this.evaluate();
            return;
        }
        else {
            let value = this.processDigit(char, this.current);
            if (value !== undefined) {
                this.current = value;
                return;
            }
            else {
                let value = this.processOperator(char);
                if (value !== undefined) {
                    this.evaluate();
                    this.operator = value;
                    return;
                }
            }
        }
        throw new Error(`Unsupported input: '${char}'`);
    }

    public getResult() {
        return this.memory;
    }
}
```

```
export function test(c: Calculator, input: string) {
    for (let i = 0; i < input.length; i++) {
        c.handleChar(input[i]);
    }

    console.log(`result of '${input}' is '${c.getResult()}'`);
}
```

Here is a simple test for the calculator using the exposed `test` function.

TestCalculator.ts

```
import { Calculator, test } from "./Calculator";

let c = new Calculator();
test(c, "1+2*33/11="); // prints 9
```

Now to extend this to add support for input with numbers in bases other than 10, let's create `ProgrammerCalculator.ts`

ProgrammerCalculator.ts

```
import { Calculator } from "./Calculator";

class ProgrammerCalculator extends Calculator {
    static digits = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B", "C", "D", "E", "F"];

    constructor(public base: number) {
        super();
        const maxBase = ProgrammerCalculator.digits.length;
        if (base <= 0 || base > maxBase) {
            throw new Error(`base has to be within 0 to ${maxBase} inclusive.`);
        }
    }

    protected processDigit(digit: string, currentValue: number) {
        if (ProgrammerCalculator.digits.indexOf(digit) >= 0) {
            return currentValue * this.base + ProgrammerCalculator.digits.indexOf(digit);
        }
    }
}

// Export the new extended calculator as Calculator
export { ProgrammerCalculator as Calculator };

// Also, export the helper function
export { test } from "./Calculator";
```

The new module `ProgrammerCalculator` exports an API shape similar to that of the original `Calculator` module, but does not augment any objects in the original module. Here is a test for our `ProgrammerCalculator` class:

TestProgrammerCalculator.ts

```
import { Calculator, test } from "./ProgrammerCalculator";

let c = new Calculator(2);
test(c, "001+010="); // prints 3
```

Do not use namespaces in modules

When first moving to a module-based organization, a common tendency is to wrap exports in an additional layer of namespaces. Modules have their own scope, and only exported declarations are visible from outside the module. With this in mind, namespaces provide very little, if any, value when working with modules.

On the organization front, namespaces are handy for grouping together logically-related objects and types in the global scope. For example, in C#, you're going to find all the collection types in `System.Collections`. By organizing our types into hierarchical namespaces, we provide a good "discovery" experience for users of those types. Modules, on the other hand, are already present in a file system, necessarily. We have to resolve them by path and filename, so there's a logical organization scheme for us to use. We can have a `/collections/generic/` folder with a list module in it.

Namespaces are important to avoid naming collisions in the global scope. For example, you might have `My.Application.Customer.AddForm` and `My.Application.Order.AddForm` – two types with the same name, but a different namespace. This, however, is not an issue with modules. Within a module, there's no plausible reason to have two objects with the same name. From the consumption side, the consumer of any given module gets to pick the name that they will use to refer to the module, so accidental naming conflicts are impossible.


For more discussion about modules and namespaces see [Namespaces and Modules \(./namespaces-and-modules.html\)](#).

Red Flags

All of the following are red flags for module structuring. Double-check that you're not trying to namespace your external modules if any of these apply to your files:

- A file whose only top-level declaration is `export namespace Foo { ... }` (remove `Foo` and move everything 'up' a level)
- A file that has a single `export class` or `export function` (consider using `export default`)
- Multiple files that have the same `export namespace Foo {` at top-level (don't think that these are going to combine into one `Foo` !)

Made with ❤ in Redmond Follow @Typescriptlang (<https://twitter.com/typescriptlang>)

Privacy (<https://go.microsoft.com/fwlink/?LinkId=521839>) ©2012-2018 Microsoft  Microsoft