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](https://www.typescriptlang.org/docs/handbook/modules.html#export). 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](https://www.typescriptlang.org/docs/handbook/modules.html#import).

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** numberRegexp = /^[0-9]+$/;

**export** **class** ZipCodeValidator **implements** StringValidator {

isAcceptable(s: string) {

**return** s.length === 5 && numberRegexp.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 && numberRegexp.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** numberRegexp = /^[0-9]+$/;

**export** **default** **function** (s: string) {

**return** s.length === 5 && numberRegexp.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** numberRegexp = /^[0-9]+$/;

**class** ZipCodeValidator {

isAcceptable(s: string) {

**return** s.length === 5 && numberRegexp.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** numberRegexp = /^[0-9]+$/;

**export** **class** ZipCodeValidator **implements** StringValidator {

isAcceptable(s: string) {

**return** s.length === 5 && numberRegexp.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" } ${ name }`);

}

});

Optio