Modules

- In TypeScript you can use modules to organize your code, avoid polluting the global space, and expose functionalities for others to use.
- Multiple modules can be defined in the same file. However, it makes more sense to keep on module per file
- If you want, you can split a single module across multiple files
- If you decide to split a module across different files, this is how you would do it:
  - Create the module file: `mymodule.ts` and declare your module there: `module MyModule {}`
  - Create another file: `mymodule.ext1.ts` and on top of the file add: `/// <reference path="mymodule.ts" />`. Then in the file, you can use the same name of the module and add more stuff to it: `module MyModule { // other stuff... }`
  - Then in your main file, you need two things on top of the file:
    - `/// <reference path="mymodule.ts" />`
    - `/// <reference path="mymodule.ext1.ts" />`
  - Then, you can use the name of your module to refer to the symbols defined: `MyModule.something`, `MyModule.somethingElse`
- TypeScript has two system: one used internally and the other used externally
- External modules are used if your app uses CommonJS or AMD modules. Otherwise, you can use TypeScript's internal module system
- Using TypeScript's internal module system, you can:
  - use the `module` keyword to define a module: `module MyModule { ... }`
  - split modules into different files that contribute to a single module
  - use the `/// <reference path="File.ts" />` tag to tell the compiler how files are related to each other when modules are split across files
- Using TypeScript's external module system:
  - you cannot use the `module` keyword. The `module` keyword is used only by the internal module system.
  - instead of the `reference` tag, you can use the `import` keyword to define the relationship between modules
  - you can import symbols using the file name: `import mymodule = require('mymodule')`

Simple Module

Let's create a simple module that contains two classes. The first class is a vehicle class and the second is a car class that inherits from the vehicle class. Then we are going to expose the car class to the outside world and import it from another file. The project files for this section are in `angular2-intro/project-files/basic-module`.

First, create the `main.ts` file and copy paste the following:

main.ts

```
1 module MyModule {
2   class Vehicle {
3     constructor (public name: string = 'Vehicle', private _distance: number =
4     get distance():number { return this._distance; }
5     set distance(newDistance: number) { this._distance = newDistance; }
6     move() { this.distance += 1 }
7   }
8 }
```

- On line 1 we are defining the module called `MyModule`.
- Inside this module we have defined a class called `Vehicle` that has a distance property and a setter and getter.

Now we want to create a class and export it so that it can be imported by others:

main.ts

```
1 module MyModule {
2   class Vehicle {
3     constructor (public name: string = 'Vehicle', private _distance: number =
4     get distance():number { return this._distance; }
5     set distance(newDistance: number) { this._distance = newDistance; }
6     move() { this.distance += 1 }
7   }
8   // -> adding the car class
```

```
 9    export class Car extends Vehicle {
10      constructor (public name: string = 'Car') {
11        super();
12      }
13    }
14  }
```

- On line 9 we are using the `export` keyword to indicate that the `Car` class is exposed and can be used by others.

Now, let's create a car using the `Car` class defined in the `MyModule` module:

```
1  const mycar = new MyModule.Car('My Car');
2  console.log(mycar.name);
```

Note that we accessed the `Car` class using the `MyModule` symbol: `MyModule.Car`. Now we can split up the module into its own file and import it into the main file. Let's create a file called `MyModule.ts` and move the module definition to that file. Now in our main file we are just going to import the module and use the car class from it.

`main.ts`

```
1  /// <reference path="MyModule.ts" />
2  const mycar = new MyModule.Car('My Car');
3  console.log(mycar.name);
```

Note that we can create an alias to the `MyModule` using `import AliasName = MyModule`. Now you can reference the module name with `AliasName`:

```
1  /// <reference path="MyModule.ts" />
2  import AliasName = MyModule;
3  const mycar = new AliasName.Car('My Car');
4  console.log(mycar.name);
```

Now if we run this in debug mode, the compiler will complain that it can't find the `MyModule` reference. Because of that we need to make some changes to our config files. First, we are going to add the `out` property in the `tsconfig.json` file. This will tell the compiler to compile all the files into a single file:

```
"out": "output/run.js",
```

So our `tsconfig.json` file will look like this:

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "module": "commonjs",
    "target": "es5",
    "sourceMap": true,
    "outDir": "output",
    "out": "output/run.js",
    "watch": true
  }
}
```

Now if you run the build, you should see that all the project has been compiled into `output/run.js`. In addition to the `tsconfig.json` file, we are going to update the `launch.json` file and add a new configuration field:

```
{
  "name": "TS All Debugger",
  "type": "node",
  "program": "output/run.js",
  "stopOnEntry": false,
  "sourceMaps": true
}
```

Now we should be able to use the debugger and put breakpoints in our TypeScript files. Select `TS All Debugger` from the debugger dropdown and run the debugger and it should stop if you put a breakpoint in any of your TypeScript files.

NOTE Using the configuration files above we can compile all the TypeScript files into a single JavaScript file. But sometimes that is not what you want. Be aware that using the above

configuration you will not get an output for each TypeScript file.

Splitting Internal Modules

Internal modules in TypeScript are open ended. This means that you can define a module with the same name in different files and keep adding to it. This is also known as merging. In this section we are going to demonstrate merging multiple files that contribute to a single module called `Merged`. The project files for this section are in `angular2-intro/project-files/merged-module`

First, we are going to make two files: `A.ts` and `B.ts`. In each file we are going to define the `Merged` module:

```
// A.ts
module Merged {
  const name = 'File A'; // not exported
  export class Door {
    constructor (private _color = 'white') {}
    get color() { return this._color; }
    set color(newColor) { this._color = newColor; }
  }
}
```

and then the `B.ts` file:

```
// B.ts
module Merged {
  const name = 'File B'; // not exported
  export class Car {
    constructor(public distance = 0) {}
    move () {this.distance += 1;}
  }
}
```

We just created two files called `A.ts` and `B.ts` and each file we defined the `Merged` module and added a class to each and exported it. Now we are going to make the `main.ts`

file and reference these two files:

```
// main.ts
/// <reference path="./A.ts" />
/// <reference path="./B.ts" />
```

And now we can use the classes defined in the `Merged` module, that is the `Car` and the `Door` class:

```
/// <reference path="./A.ts" />
/// <reference path="./B.ts" />
const car: Merged.Car = new Merged.Car();
const door: Merged.Door = new Merged.Door();
door.color = 'blue';
car.move();
car.move();
console.log(car.distance);
console.log(door.color);
```

if you run the build task (command + shift + b) and hit F5 you should see the following output:

```
node --debug-brk=19237 --nolazy output/run.js
Debugger listening on port 19237
2
blue
```

External Modules

In addition to TypeScript's internal module system, you can use external modules as well. In this section we are going to demonstrate how you can use external modules in TypeScript. The project files for this section are in `angular2-intro/project-files/external-module`

Let's say I have a JavaScript Node module defined in CommonJS format in a file called `common.js` :

```
// common.js
module.exports = function () {
  this.name = 'CommonJS Module';
};
```

In order to import this we need to do two things: first, we need to install Node's Type Definitions. Then we need to require the module. To install Node's Type Definitions run the following the terminal in the root of your project:

```
tsd install node --save
```

Now you should see a folder called `typings` containing the type definitions. Now that we have Node's type definitions, let's add a reference to it on top of `main.ts`:

```
// main.ts
/// <reference path="./typings/node/node.d.ts" />
```

and then we are going to require the module and log it to the console:

```
// main.ts
/// <reference path="./typings/node/node.d.ts" />
const common = require('./common');
console.log(common()); // --> CommonJS Module
```

After running the build task (command + shift + b), and running the file (F+5) you should see the following output:

```
node --debug-brk=32221 --nolazy run.js
Debugger listening on port 32221
CommonJS Modules
```

Note the configuration files that we are using:

`tsconfig.json`

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "module": "commonjs",
    "target": "es5",
    "sourceMap": true,
    "outDir": "output",
    "out": "run.js",
    "watch": true
  }
}
```

launch.json

```
{
  "version": "0.1.0",
  "configurations": [
    {
      "name": "TS All Debugger",
      "type": "node",
      "program": "./run.js",
      "stopOnEntry": false,
      "sourceMaps": true
    }
  ]
}
```