
3

Modules and Namespaces

Modules, Exports and Imports, Path mapping, Namespaces



Global and Local scope

- Scope refers to the current context of code
- In JS/TS there are two types of scopes:
- **Global scope**
 - Any variable, function or a block can be accessed from another script file
- **Local scope**
 - Variables are scoped to the file and are not accessible from other files



Modules

- A feature of ECMAScript standard
- A module is a file that encapsulate and exports pieces of code
- Create a local scope to variables , functions, ...etc
- All variables, classes, functions ...etc, are **NOT VISIBLE** outside the module
- Need for explicit **EXPORTS** and **IMPORTS**



Modules

WITHOUT modules

Script

```
FuncA() {  
  ...  
}  
  
FuncB() {  
  ...  
}  
  
FuncC() {  
  ...  
}
```

All code in a single file

WITH modules

Script

```
import FuncA  
import FuncB  
import FuncC  
  
...
```

Code splitted into
multiple modules

Module

```
export FuncA() {  
  ...  
}
```

Module

```
export FuncA() {  
  ...  
}
```

Module

```
export FuncA() {  
  ...  
}
```



Export

- Exporting a declaration from a module makes it visible for other modules (files)
- Any declaration can be exported using **export** keyword

```
//ZipCodeValidator.ts  
export const numberRegexp = /^[0-9]+$/;  
  
export class ZipCodeValidator implements StringValidator {  
  
}
```



Export

- It's also possible to export declarations using export statements

```
export { ZipCodeValidator };  
export { ZipCodeValidator as mainValidator };
```

- To export a single value or have fallback value of a module, use **default** export

```
export default function cube(x) {  
    return x * x * x;  
}
```



Import

- To use (reference) a declaration from another module, use import statements:
- Importing a single item:

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

- Importing the entire module (visible members only):

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



Module resolution

- The process the compiler uses to figure out what an import refers to
- Two strategies:
- **Relative imports**
 - Starts with (`/`), (`./`), (`../`) → Files of same project

```
import Header from "../components/Header";
```

- **Non-Relative imports**
 - Any other import is considered non relative → External dependencies

```
import * as $ from "jquery";
```




Path mapping

- Path mapping allows to give relative paths aliases
- Helps to have consistent paths and makes restructuring easy

`@components` \longrightarrow `src/pages/components`

`@data` \longrightarrow `src/data`

- Path mapping is **specific to Typescript**



Path mapping

```
// Import from relative paths  
import { UserComponent } from '../components/user.component';  
import { UserService } from '../../../data;
```



```
import { UserComponent } from '@components/user.component';  
import { UserService } from '@data;
```



Path mapping

- Use `tsconfig.json` compiler options to define mappings
 - Set **paths** to setup mappings between an alias and a relative path
 - Set **baseUrl** to tell the compiler where the resolution starts from

```
"compilerOptions": {  
  "baseUrl": "./src",  
  "paths": {  
    "@components/*": ["components/*"],  
    "@data/*": ["data/*"],  
  }  
}
```



Namespaces

- Namespaces are a typescript-specific way to organize code
- Namespaces are just javascript objects in the global scope
- Allows to create organization units for values, functions, classes, ..., etc
- Use `namespace` keyword to declare a namespace

```
namespace <namespace_name> {  
    export var variable ...  
    export class A {  
    }  
}
```



Namespaces

- Use export keyword to export members from a namespace
- To reference a namespace from another file, use the triple slash reference syntax

```
/// <reference path="Namespace_Path.ts" />
```

- To access declaration, use the same syntax as property access for objects

```
namespaceName.className;  
namespaceName.functionName;
```



Namespaces VS Modules

Namespaces

- Must use the namespace keyword
- Used for logical grouping with locale scoping
- Must use the triple slash reference syntax
- No need for module loader

Modules

- No keyword, a file is a module
- Used to organize code in separate files
- Must use import statements
- Must include the module loader