Advanced TypeScript Part - 3

Ashutosh Sarangi

NeoSOFT® TECHNOLOGIES

www.neosofttech.com

Agenda

- Modules
- Namespaces
- Namespaces Vs Modules
- Module Resolution
- Declaration Merging
- MCQ

Modules

- files containing a top-level export or import are considered modules.

Q. Before Modules What was the Scenario ?

Importing the Entire Module into a Variable:-

```
import * as Emp from "./Employee"
```

Renaming an Export from a Module:

```
import { Employee as Associate } from "./Employee"
```

Namespaces

- The namespace is used for logical grouping of functionalities. A namespace can include interfaces, classes, functions and variables to support a single or a group of related functionalities.

```
namespace <name>
{

}
```

Namespaces

Splitting Across Files

file1.ts

```
namespace Demo {
    export const x = 1;
}
```

file2.ts

```
namespace Demo {
    export const y = 2;
}
```

Test.ts

```
/// <reference path="file1.ts" />
/// <reference path="file2.ts" />

console.log(Demo.x + Demo.y);
```

Namespaces Vs Modules

NameSpace	Module
Must use the namespace keyword and the export keyword to expose namespace components.	Uses the export keyword to expose module functionalities.
Used for logical grouping of functionalities with local scoping.	Used to organize the code in separate files and not pollute the global scope.
To use it, it must be included using triple slash reference syntax e.g. ///<reference path="path to namespace file" />.	Must import it first in order to use it elsewhere.

Module Resolution

Module resolution is the process used by compiler to figure out imports.

Ex:- import {a} from ‘ModuleA’;

The compiler will also check the usage of ‘a’ through out application. If require then the compiler will check the definition of the ModuleA.

Type Of Imports

1. Relative Imports
2. Non- Relative Imports

Module Resolution

Types Of Imports

Relative Imports

It is one of the importing mechanisms. That starts with ‘/, ./, ../‘;

Ex:-

 Import {AA} from ’../service.ts’;

Non- Relative Imports

Ex:-

import {Component} from ‘@angular/core’;

Module Resolution

Types of Resolution Strategy

1. Classic
2. Node

Classic - Relative

Import {B} from ‘ModuleB’;

LookUps:-

/root/src/folder/ModuleB.ts
/root/src/folder/ModuleB.d.ts

Module Resolution

Classic- Non-Relative

Import {B} from ‘ModuleB’;

LookUps:-

/root/src/folder/ModuleB.ts
/root/src/folder/ModuleB.d.ts
/root/ModuleB.ts
/root/ModuleB.d.ts
/ModuleB.ts
/ModuleB.d.ts

Module Resolution

Node

In this resolution strategy attempts node.js module resolution mechanism at run time.

Using require node.js imports the modules

The require behaviour is different from relative and non-relative imports in different manner.

Module Resolution

Node - Relative

Var x = require ('./moduleB');

As a file

/root/src/moduleB.js

As A Folder —> It specifies mainModule

/root/src/moduleB/lib/mainModule.js

As a folder (index.js) —> implicitly consider that folder is main Folder

/root/src/moduleB

Module Resolution

Node- Non-Relative

Node will look for your module in specific Node_module Folder.

Resolution Flags

1. Base URL
2. Path Mapping
3. Virtual Directories

Module Resolution

Base URI:-

Setting Base URL informs the Compiler where to find Modules

```
Comileroptions:{
	‘baseUrl’: ‘.’
}
```

Path Mapping

Some Modules are not located Under Base Url (JQuery)

```
compilerOptions:{
‘baseUrl’:’’,
‘paths’:{
	‘jquery’:[‘node_modules/jquery/dist’]
}
```

Module Resolution

Virtual Directories

```
compilerOptions:{
rootDirs:[
        ‘src/views’,
        ‘generated/templete/Views’
]
```

Declaration Merging

Declaration merging is when the TypeScript complier merges two or more types into one declaration provided they have the same name.

TypeScript allows merging between multiple types such as interface with interface, enum with enum, namespace with namespace, etc.

Note:-

notable merge that isn’t permitted is class with class merging.

Declaration Merging

Let’s get started with interface with interface merging by looking at an example:

```
interface Person {
  name: string;
}

interface Person {
  age: number;
}

interface Person {
  height: number;
}
```

```
class Employee implements Person
{
  name = "Mensah"
  age = 100;
  height = 40
}

const employee = new Employee();
console.log(employee) //
```

Declaration Merging

If any of the interfaces to be merged contain the same property name and that property isn't a function, then the type of the properties must be the same or else the complier will throw an error.

```
interface Person {
  name: string;
  zipCode: string;
}

interface Person {
  age: number;
  zipCode: string; // acceptable
}
```

```
interface Person {
   zipCode: number; // error
}
```

Declaration Merging

When the elements in the merged interfaces are functions and they have the same name, they are overloaded, that is, depending on the type of argument passed, the appropriate function will be called.

```
interface Person {
  speak(words: string);
}
interface Person {
  speak(words: number);
}
const person: Person = {
  speak: (wordsOrNum) => wordsOrNum
}
```

```
console.log(person.speak("Hi"))
console.log(person.speak(2))
```

Declaration Merging

Priority:-

```
interface Person {
  speak(words:string);
}
interface Person {
  speak(words: any);
}
interface Person {
  speak(words: number);
  speak(words: boolean);
}
```

merged interface looks like

```
interface Person {
 // functions in the last interface appear at the top
  speak(words: number);
  speak(words: boolean);

  // function in the middle interface appears next
  speak(words: any):number;

  // function in the first interface appears last
  speak(words: string):string;
}
```

Declaration Merging

```
interface Person {
  speak(words: number);
  speak(words: "World!"); // string literal type
}

interface Person {
  speak(words: "Hello"); // string literal type
}

interface Person {
  speak(words: string);
}
```

MCQ

1. Difference Between *.d.ts vs *.ts ? (interface)

```
interface test{
  name: string;
}

let a: test = {
  name: 'Ashu'
}
```

How the Above code look like in .js File

THANK YOU

NeoSOFT®
TECHNOLOGIES

www.neosofttech.com