upGrad

Introduction to TypeScript

Course: Introduction to TypeScript

Lecture on: Modules in TS

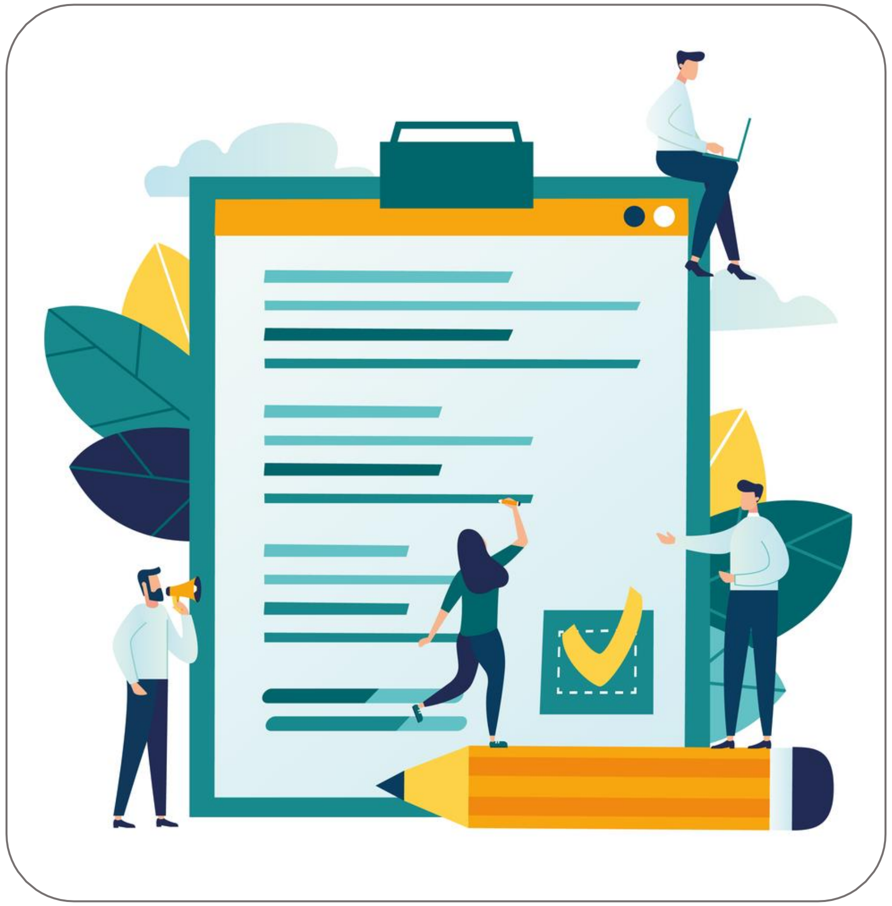
Instructor: Aquib Ajani

COURSE ROADMAP

- Introduction to TypeScript (TS)
- Operators, Conditions and Loops In TS
- Functions in TS
- Hands-On Coding Session - I
- Classes and Interfaces in TS
- Type Manipulation in TypeScript
- **Modules in TS**
- Hands-On Coding Session - II

TODAY'S AGENDA

- Introduction to Modules
- How to Declare Modules
- Import/Export
- Module Resolution



Modules in TS

MODULES IN TS

Introduction to Modules

Modules:

- The concept of modules in JS is also shared by TypeScript. Any file having top-level import or export is called a module. Modules are executed within their own scope and not in the global scope

There are a few things that need to be kept in mind before creating modules:

- Syntax: How to use import and export
- Module path resolution: How the paths are defined for the modules and their nesting
- Output: How the emitted JS module looks

MODULES IN TS

Introduction to Modules

Advantages of Modules:

- It is very simple to start using modules with TypeScript.
- A few advantages of modules are as follows:
  - Release cleaner applications and help understand dependency between components in a better way
  - Help when structured properly for better debugging and maintenance
  - Encapsulate various logic related to different functionality or screens that can be easily identified

MODULES IN TS

Introduction to Modules

How to declare modules:

The main module can be exported like:

```
//@filename: demo.ts
export default function demo() {
  console.log(“Demo module");
}

you can then import like:
import demo from “./demo.js";
demo();

Also can have multiple exports :
export var pi = 3.14;
export let squareTwo = 1.41;
export const phi = 1.61;
```

MODULES IN TS

Export

Any type of declaration can be exported by adding the 'export' keyword, whether that may be a variable, function, class, interface, etc.

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

import { StringValidator } from "./StringValidator";

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

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

MODULES IN TS

Export

Export Statements:

- When you need to rename exports, export statements can be used.

  Example:

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

MODULES IN TS

Export

Re-Exports:

- Modules are often extended by other modules and partially expose some of their features. A re-export does not import it locally or introduce a local variable.

  The previous example can be written as follows:

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

MODULES IN TS

Export

Default Exports:

- Default exports are marked by the keyword ‘default’, and each module can export optionally default export

Example:

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

In import file:

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

MODULES IN TS

Export

Export all as x:

- To re-export another module with a name, you can use shorthands like export * as ns

  Example:

```
export * as utilities from "./utilities";
```

  This takes all of the dependencies from a module and makes them an exported field; you can import it like this:

```
import { utilities } from "./index";
```

MODULES IN TS

Import

Importing a single module:

- Import works the same way as export. It is done using the keyword ‘import’

  Example:

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

let myValidator = new ZipCodeValidator();

imports can also be renamed
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
let myValidator = new ZCV();
```

MODULES IN TS

Export

- Importing the entire module into a single variable:
- Import the entire module can be done using the “*” keyword

Example:

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

Importing a module for side effects only:

Modules can be imported directly using a global state:

Example:

```
import “./my-module.js";
```

MODULES IN TS

Import

Importing Types:

- Types can be imported using import type or import statement

  Example:

```
// Re-using the same import
import { APIResponseType } from "./api";

// Explicitly use import type
import type { APIResponseType } from "./api";
```

MODULES IN TS

Module Resolution

- When a compiler wants to find out what an import refers to, it is called module resolution.
- For example:

  Import { a } from “moduleA”;
- Here, the compiler tries to find out the file (.ts/.tsx/.d.ts) depending on the code

MODULES IN TS

Module Resolution

- Relative vs non-relative module imports:
- Based on the reference, the module can be imported relatively or non-relatively.
- A relative import starts with /, ./, ../

  Example:

```
import Entry from "./components/Entry";
import { DefaultHeaders } from "../constants/http";
import “/mod";
```

- Others are non-relative:

  Example:

```
import * as $ from "jquery";
import { Component } from "@angular/core";
```

MODULES IN TS

Module Resolution

- Module resolution strategies:
- There are two main ways of module resolution.

Classic:

- This used to be the default strategy for TS, but now, it is majorly used for backward compatibility. A relative import is solved using the relative directory to the current file.

So, import { b } from "./moduleB"

In source file

```
/root/src/folder/A.ts
```

Result would reside in the following files:

```
/root/src/folder/moduleB.ts
/root/src/folder/moduleB.d.ts
```

MODULES IN TS

Module Resolution

- For non-relative imports, the compiler directly checks the directory containing the importing file and looks for the matching file.
- A non-relative import to moduleB such as **import { b } from "moduleB"**, in a source file **/root/src/folder/A.ts,** would result in attempting the following locations for locating **“moduleB":**

```
/root/src/folder/moduleB.ts
/root/src/folder/moduleB.d.ts
/root/src/moduleB.ts
/root/src/moduleB.d.ts
/root/moduleB.ts
/root/moduleB.d.ts
/moduleB.ts
/moduleB.d.ts
```

MODULES IN TS

Module Resolution

Node:

- As the name suggests, it follows the nodejs module resolution strategy.
- In nodejs, it is done using the require function passing up the relative or relative path, and this function works differently for relative and non-relative paths when passed

  Example:

  Assuming the import is like:

  Var x = require("./moduleB")
  Ask the file named /root/src/moduleB.js whether it exists
- Ask the folder /root/src/moduleB if it contains a file named package.json that specifies a 'main' module. In our example, if Node.js found the file /root/src/moduleB/package.json containing { "main": "lib/mainModule.js" }, then Node.js will refer to /root/src/moduleB/lib/mainModule.js
- Ask the folder /root/src/moduleB if it contains a file named index.js. This file is implicitly considered the folder's 'main' module
- TS uses the same strategy to find the located file from which the module is imported using package.json and then for the index.js

MODULES IN TS

Module Resolution

Tracing module resolution:

The compiler can visit files outside the current folder when resolving a module. It becomes hard when the module is not resolved, and enabling compiler module resolution tracing using traceResolution provides insight into what happened during the module resolution process.

Things to look out for are as follows:

-Name and location of the import

======== Resolving module **'typescript'** from **'src/app.ts'**. ========

- The strategy the compiler is following

Module resolution type is not specified using **'NodeJs'**.

- Loading of types from npm packages

'package.json' has **'types'** field './lib/typescript.d.ts' that references 'node_modules/typescript/lib/typescript.d.ts'

-Final result

==== Module name 'typescript' was **successfully resolved** to 'node_modules/typescript/lib/typescript.d.ts'. ===

MODULES IN TS

Module Resolution

—-noResolve

- Normally, the compiler will attempt to resolve all module imports before it starts the compilation process. Every time it successfully resolves an import to a file, the file is added to the set of files the compiler will process later.
- The noResolve compiler option instructs the compiler not to ‘add’ any files to the compilation that were not passed on the command line. It will still try to resolve the module to files, but if the file is not specified, it will not be included

```
import * as A from "moduleA"; // OK, 'moduleA' passed on the command-line
import * as B from "moduleB"; // Error TS2307: Cannot find module 'moduleB'.
tsc app.ts moduleA.ts --noResolve
```

KEY TAKEAWAYS

- Introduction to modules
- How to declare modules
- Import/Export
- Module resolution

TASKS TO COMPLETE AFTER THE SESSION

MCQs
Coding Questions

upGrad

Thank you!