Lab 3: Modules and Namespaces

This lab outlines the various ways to organize your code using modules and namespaces in TypeScript.

Modules

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. Modules are declarative; the relationships between modules are specified in terms of imports and exports at the file level.

Global scope

First, we'll separate our code into multiple files but still execute it in the global scope:

1. Open the starter project with VSCode
2. Create a new file `models.ts` in the `src` folder
3. Move all the types (interface or type) into `models.ts`
4. Create a new file `data.ts` in the `src` folder
5. Move `tasks` variables and it's dependicies (`me` and `task`) into `data.ts`
6. Move the CRUD functions (create, update and delete) into `data.ts`
7. Run `tsc` command to type-check

> Altough we sperated our elements (types, variables and functions) into separate files, the TS show no error and that's because no file have an import statement and as result everything is considered in the **global scope**.

Exporting

Once we use an export , we define our module that is scoped to the file:

1. Open `models.ts` file and add an export eyword for every type declaration, example:

```
export type Tasks = Array<Task>
```

2. Open `data.ts` file and add an export eyword for every CRUD function , example:

```
export function deleteTask(id: number): boolean {
    //...
}
```

3. In the same file (`data.ts`) export the `tasks` variable as a **default export** by adding this statement in the end of the file

```
export default tasks;
```

> After adding `export` keyword , each file is considered as a module and have it's own local scope, therefore we'll need to add `import` statements to use the exports.

Importing

To use exports, we must reference them using imports statements :

1. In the data file (`data.ts`), fix the errors by adding the missing import statements like the following

```
import { Person, Task, Tasks } from  "./models"
```

2. In the main file (`main.ts`), import the tasks variable from `data.ts` file

```
import tasks from  "./data"
```

> `tasks` is a default export, so you can rename it when importing it , for example this is also correct : `import allTasks from "./data"`

3. Compile using `tsc`

Path mapping

Typescript support path mapping which gives a path (folder or file) an alias that can be used within the imports statements:

1. Open TS configuration file : `tsconfig.json`
2. Search for the property `baseUrl`, uncomment it and set it to the root project directory (`./`)

```
"baseUrl": "./",
```

3. Uncomment the `paths` property and add the following mappings:

```
"paths": {
    "@data":["./src/data"],
    "@models":["./src/models"],
},
```

> By defiing paths, `@data` is resolved to `./src/data` relatively to the baseUrl which is the root directory

4. Import all the relative paths to use path mapping instead **main.ts**

```
import tasks from  "@data"
```

data.ts

```
import { Person, Task, Tasks } from  "@models"
```

5. Run `tsc` and check for errors.

Namespaces

Namespaces are a TypeScript-specific way to organize code. Namespaces are simply named JavaScript objects in the global namespace. This makes namespaces a very simple construct to use to keep track of our types and not worry about name collisions with other objects.

1. Head over `models.ts` file and create a namespace `Models`

```
namespace Models {

}
```

2. Move all the types into `Models` namespace

```
namespace Models {
    export  type Tasks ...

    export  interface Person ...

    export interface Task ...
}
```

3. Go to `data.ts` , remove module import statement and replace it by the triple slash reference syntax used by namespaces

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

4. Fix all the type annotations in the file, example:

```
//Using module
const tasks: Tasks = [task]

//Using namespace
const tasks: Models.Tasks = [task]
```

5. Run `tsc` and check for errors.

Use NPM dependencies

Any JS or TS project (Web, Mobile or Backend) needs to use external dependencies, for JS/TS external dependencies are manager by the NPM registry (Node Package Manager). Open source developers and developers at companies use the npm registry to contribute packages to the entire community.

In this section, you'll learn how to setup your project to use npm in order to install dependencies and use them:

1. Run the following command to initialize the project

```
npm init
```

> After running this command, a new file `package.json` should be created, containing the project description and list of NPM dependencies

2. To install a NPM dependency, use the following command

```
npm install --save <name-of-dependency>
```

or

```
yarn add <name-of-dependency>
```

- Run the following command to install Moment.js for date formatting

```
npm install --save moment
```

> After running the previous command, a new folder `node_modules` is created, which contains the dependencies bundles, and a new line in the `dependencies` section in `package.json` file

```
"dependencies": {
    "moment": "^2.29.4"
}
```

4. Run the following command to install a dev dependency

```
npm install --save-dev tsc-alias
```

> A dev-depdendency is a dependency which is used only at design (devlopment) time and is not included in the runtime.

2. Add a command shortcut in `scripts` section in `package.json` file

```
"scripts": {
    "start": "tsc && tsc-alias && node dist/main.js",
    "test": "echo \"Error: no test specified\" && exit 1"
}
```

> The `start` command shortcut, runs `tsc` to type check then invoke `tsc-alias` to replace path aliases by the real realtive paths and finaly run the output file `main.js`

3. To run commands that are specified in `scripts`, use `npm <name>` for example:

```
npm start
// or yarn start
```