

# Namespaces and Modules



**SoftUni Team**  
**Technical Trainers**



**SoftUni**



**Software University**

<https://about.softuni.bg>

sli.do

**#typescript**

1. Introduction to Namespaces
2. Introduction to Modules
  - Exporting and Importing
3. Namespaces vs Modules

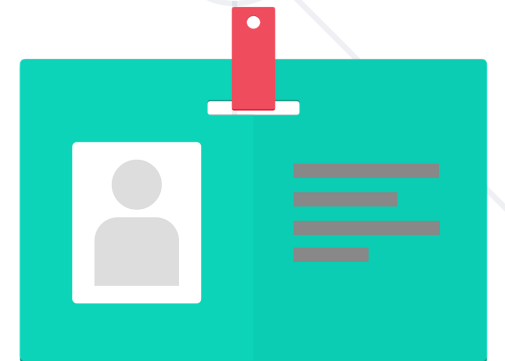




# **Intro to Namespaces**

# Definition

- Namespaces are used to **logically grouped** functionalities
- Previously referred as **internal modules** in TypeScript
- Defined with **namespace** keyword
- Namespaces may include **functions, classes, interfaces** and **variables**



# Access

- The elements of the namespace that must be accessed from the outside must be marked with **export** keyword
- In order to access namespaces from different files we must use the reference syntax  
**///  
reference path = "file.ts" />**



# Example: Namespace

```
namespace printMessages {  
  export function messenger(message: string | string[])  
  {  
    return `${message}`;  
  }  
  export interface meetPerson {  
    meetPerson(): string  
  }  
}  
console.log(printMessages.messenger('Hello')); //Hello
```

namespace declaration

export to use the  
interface outside

# Multiple Files Namespaces


- In order to access namespaces from different files we must use the reference syntax  
**///**
- In order to compile the file we must
  - Compile the ts file - **tsc fileName.ts**
  - Use the outFile - **tsc --outFile fileName.js fileName.ts**
  - Compile the js file - **node fileName**





# Aliases

- Used to simplify the work with namespaces
- Used with **import** keyword
- Often used as nested namespaces



```
namespace Shops {  
  export namespace TechStores {  
    export class PCStore {}  
    export class AudioStore {}  
    export class TVStore {}  
  }  
}  
  
--Name of file - app.ts  
import stores = Shops.TechStores;  
let pcStore = new stores.PCStore();
```

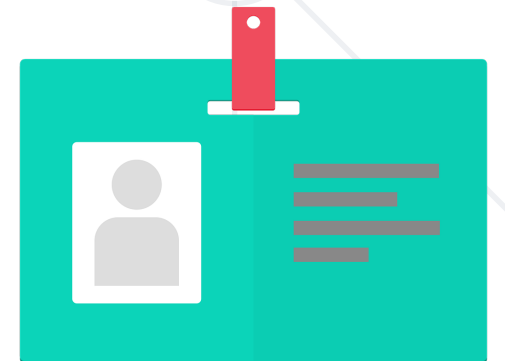


# Intro to Modules

# Definition



- Modules are executed in their **own scope**, not the global
- A **set of functions** to be included in applications
- Resolve name collisions
- In order to be accessed from the outside they need to be marked with **export** keyword



# Access

- To consume a function, class, interface or variable exported from another module we must use an **import** form
  - **import** { name } from "./location" - import specific element
  - **import** \* as variable from "./location"; - imports the entire module in single variable





# Exporting and Importing

- There are three ways to use **export** statements:
  - A: `export function numberValidation(num: number): number {...}`
  - B: `export { numberValidation };`
  - C: `export { numberValidation as isValidNum }; //isValidNum is alias`
  - D: `export default function stringValidations(string: string): string {...}`
- In cases **A** and **B** there is **no difference** rather than syntax
- There might be only one **export default** in a file

# Example: Export and Import Statements

```
--exports
export default function checkInput<T>(information: T): T {
  if (information) { return information; }
  else { throw new Error('The information passed is not valid') }
}
export function stringValidations(string: string): string {
  if (string.length > 0 && string.length <= 20) { return string; }
  else { throw new Error('String is not valid'); }
}
export function numberValidation(num: number): number {
  if (num > 0 && num <= 999) { return num; }
  else { throw new Error('Number is not valid'); }
}
export { numberValidation as isValidNum };
```

```
--Imports  
import * as validations from './validations';  
//validations is alias  
import checkInput from './validations';  
import { isValidNum } from './validations';  
  
// Some code logic
```


- In order to compile the file we must
  - Compile the ts file - **tsc fileName.ts**
  - Use the outFile - **tsc --module commonjs fileName.ts**
  - Compile the js file - **node fileName**






# Namespaces vs Modules

# Namespaces vs Modules

- 
- Namespaces: global containers for code organization
  - Enclosed using namespace keyword
  - Can be split across multiple files but combined during compilation
  - Can contain variables, interfaces, functions, classes, etc.

```
namespace Shapes {  
  export interface Circle {  
    radius: number;  
  }  
}
```

# Namespaces vs Modules

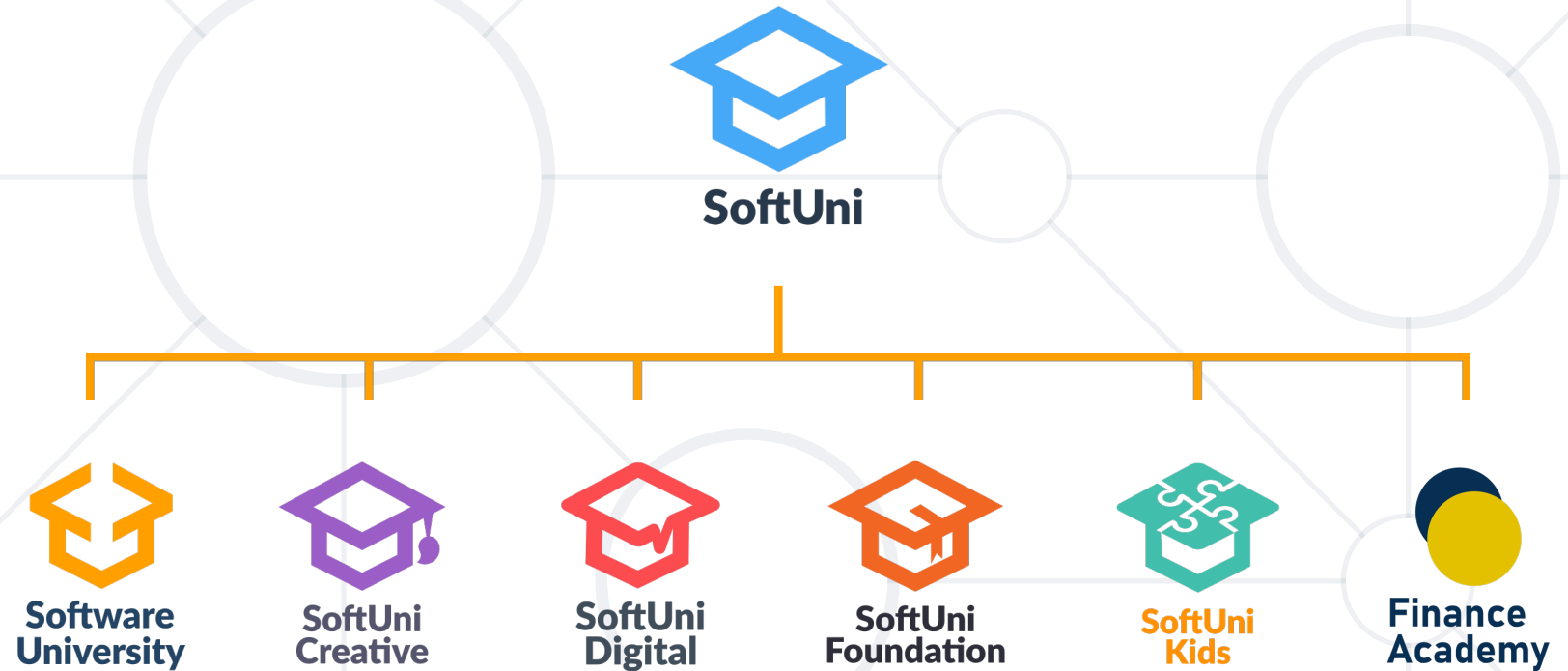
- 
- Modules: modularize code into separate file
  - Enclosed using export and import keywords
  - Are more file-based and can be loaded asynchronously
  - Can contain variables, functions, classes, etc., but not directly at the root level

```
export interface Circle {  
  radius: number;  
}  
  
import { Circle } from './circle';
```

- Namespaces are **logically grouped** functionalities
- Modules are a **set of functions** to be included in applications
- Modules do not **pollute the global scope**



# Questions?



# SoftUni Diamond Partners



- Software University – High-Quality Education, Profession and Job for Software Developers

- [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)

- Software University Foundation

- [softuni.foundation](http://softuni.foundation)

- Software University @ Facebook

- [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>

