Modularizing TypeScript

Organizing Code

Modularization

- In TypeScript (and JavaScript), any object declared outside of a function has the global scope. This means that by default, any function, class, or component in your application can access said object.
- There are two ways to organize components in your code in TypeScript
  - Namespaces (Internal Modules)
  - Modules (External Modules)
- Requires components to be exported, then imported somewhere else in the application

Namespace

Internal Modules

- A namespace define where a component of code exists
- It's like a theoretical structure that store variable, structural definitions, interfaces, classes, etc.

- Use the `namespace` keyword to define a namespace
- Use the `export` to expose a component
- Use `import` keyword to use an exposed component
- Typical usage is function scope but can be extended to file scope

```
// definite namespace
namespace Person.model{
    export interface Person{
        name: string,
        age: number
    }
}

// ... import for use
import Model = Person.model;
```


Modules

External Modules

- External Modules in TypeScript are used to specify and load dependencies between multiple **external** files.
    - Helps in modular development and keep code organized

- Use the module keyword to define a module
- Use the `export` to expose a component
- Use `import` keyword to use an exposed component

```
export interface Person{
    name: string;
    age: number;
}

// import for use
import { Person } from './Personmodel';
```

Running Typescript in Browser

… using transpiled JS code

- Option 1: use script tags to import transpiled JS files
- Option 2: use System.js to load in modules into our application. The library can be found at: https://cdnjs.com/libraries/systemjs/0.19.47
  - Then, import the app in our browser by adding the following lines into our script

```
System.defaultJSExtensions = true;
System.import(<appName>)
```

- Both options will import the application javaScript logic and run it in our browser.

Decorators

@decorators

- Decorators allow annotating or modifying classes and class members. Decorators provide a way to add both annotations and a meta-programming syntax for class declarations and members.
  - It can be attached to a class, method, or property
  - Uses form `@expression`, where `expression` is a function that will be called at runtime with information about the decorated declaration.
  - The decorator function typically changes the behaviour of the decorated component.