Modules

PROGRAMMING WITH TYPESCRIPT

Objectives

- To be aware of the JavaScript ES2015+ module syntax for importing and exporting

2

CONTINUED 2

Modules - Nomenclature

- Pre typescript 1.5 there was a concept of “internal modules” and “external modules”
- ECAMScript2015 introduced “modules” to JavaScript and so TypeScript has changed its terminology to match.
- Internal modules are now “namespaces”
- External modules are now simply “modules”

3

MODULES

```
//vehicles.ts
export interface Vehicle {
  wheels: number;
  make: string;
  model: string;
  accelerate(t:number): number;
}

export class Car implements Vehicle {
  wheels = 4;
  constructor(
    public make:string,
    public model:string)
  {}

  accelerate(time: number) {…}
}
```

Modules run in their own scope, avoiding pollution of the global scope

Only what is exported is exposed externally

Only what is imported is usable internally

Any declaration can be exported through using the export keyword

4

MODULES: EXPORT STATEMENTS

```
//vehicles.ts
interface Vehicle {
  wheels: number;
  make: string;
  model: string;
  accelerate(t:number): number;
}

class Car implements Vehicle {
  wheels = 4;
  constructor(
    public make:string,
    public model:string)
  {}
  accelerate(time: number) {…}
}

export { Vehicle };
export { Car as BaseCar };
```

Export statements can be used to export under a different name

5

MODULES: DEFAULT EXPORT

```
//vehicles.ts
export default interface Vehicle {
  wheels: number;
  make: string;
  model: string;
  accelerate(t:number): number;
}
```

```
// cars.ts
class Car implements Vehicle {
  wheels = 4;
  constructor(public make:string, public
model:string){}
  accelerate(time: number) {…}
}

export { Car as default };
```

Optionally a default export can be specified

Two syntaxes available

A file can only have 1 default export statement

6

Modules: Importing

- Importing is just as simple as exporting
- We use the import keyword with one of the following forms:
  - We can rename on import

```
import { Vehicle, Car } from './vehicles';

let myCar = new Car(`Ford`,`Fiesta`);
```

```
import { Car as BasicCar} from './vehicles';

let myCar = new BasicCar(`Ford`, `Fiesta`);
```

  - Or import the whole file!

```
import * as vehicles from './vehicles';

let myCar = new vehicles. Car(`Ford`,
`Fiesta`);
```

  - Importing the default is the simplest form

```
import Car from './vehicles';
```

7

Modules: Code Creation

- The TypeScript compiler is not a module loader. It will compile your code to whatever module format you tell it to.
- A module loader will be required to then make your code ready for deployment
  - This is what we have been using Webpack for although we have only been using a single file so far

8

Objectives

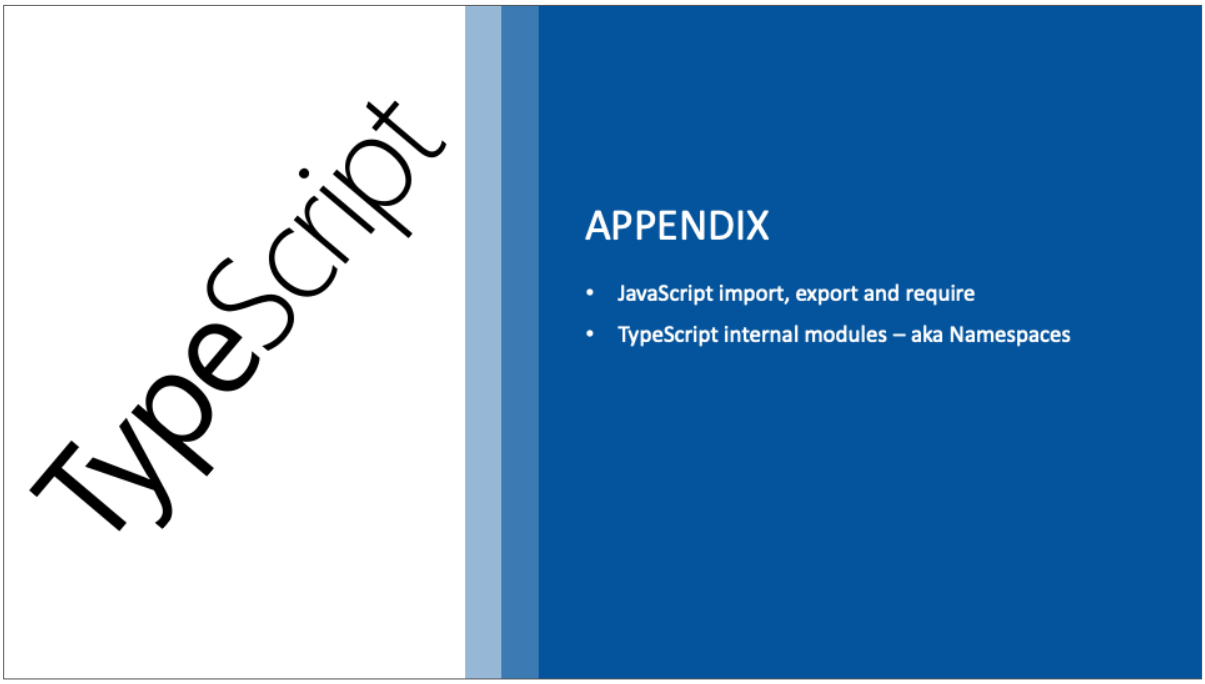
- To be aware of the JavaScript ES2015+ module syntax for importing and exporting

9

QuickLab 6 – Modules

- There is no QuickLab for this module – you will use modules in the Hackathon!

10


TypeScript
APPENDIX
• JavaScript import, export and require
• TypeScript internal modules – aka Namespaces

Modules: export = and import = require()

- The two most common module syntaxes prior to ES2015 (CommonJS and AMD) both supported the concept of an exports object. TypeScript has it's own syntax to model this workflow.

```
//vehicles.ts
class Car implements Vehicle {
  wheels = 4;
  constructor(public make:string, public model:string){}
  accelerate(time: number) {…}
}

export = Car;
```

```
//app.ts
import Car = require('./vehicles');
```

12

Namespaces - Nomenclature

- Pre typescript 1.5 there was a concept of “internal modules” and “external modules”
- ECAMScript2015 introduced “modules” to JavaScript and so TypeScript has changed its terminology to match.
- Internal modules are now “namespaces”
- External modules are now simply “modules”

13

NAMESPACES

```
//shapes.ts
namespace shapes {
  export interface Shape {
    sides: number;
    color: string;
    area: number;
  }

  export class Square implements Shape {
    sides = 4;
    constructor(
      public sideLength: number,
      public color: string
    ) {}
    get area() {
      return this.sideLength**2;
    }
  }
}

let mySquare = new shapes.Square(2, "red");

console.log(mySquare.area);
```

Namespaces provide a route for organizing our code and avoiding naming collisions arising from polluting the global scope.

Created using the namespace keyword

Public members are exposed through the export keyword

14

NAMESPACES: MULTIPLE FILES

```ts
//shapes.ts
namespace shapes {
  export interface Shape {
    sides: number;
    color: string;
    area: number;
  }
}
```

```ts
//square.ts
/// <reference path="shapes.ts" />
namespace shapes {
  export class Square implements Shape {
    sides = 4;
    constructor(public sideLength: number, public color: string)
{}
    get area() {
      return this.sideLength**2;
    }
  }
}
```

```ts
//app.ts
/// <reference path="shapes.ts" />
/// <reference path="square.ts" />
let mySquare = new shapes.Square(2, "red");

console.log(mySquare.area);
```

Usually we'd want to split our namespaces into individual files

15

Namespaces: Multiple Files Output

- TypeScript Compiler will automatically output a single file for each input file, you then need to include each in your HTML file with <script> tags.
- Alternatively you can use the –outFile option to tell the compiler to concatenate them to one file

```
tsc –outFile app.js test.ts
```

16

NAMESPACES: ALIASES

```
namespace Shapes {
  export interface Shape {
    sides: number;
    color: string;
    area: number;
  }

  export namespace Polygons {
    export class Square implements Shape {}
    export class Hexagon implements Shape {}
    export class Pentagon implements Shape {}
  }
}

import polygons = Shapes.Polygons;
let mySquare = new polygons.Square(2, "red");

console.log(mySquare.area);
```

Aliases provide a route to simplify names for commonly used objects

17