Angular 5 Training Course

Exercise B-shop

- Create a new Angular project called **shop**.

```
cd desktop
ng new shop
cd shop
ng serve -- open
```

- Simplify the component template in **src/app/component.html** so that it displays just a heading.

```
<section class="shop">
    <h1>Angular</h1>
</section>
```

Styling the template using SASS.

- SASS adds programmatic features to CSS such as variables and functions. Browsers do not recognise SASS. It needs to be transpiled back into CSS. The Angular CLI tools include support for SASS transpilation.
- We will style the template with **SASS**.
- Rename file **src/app/component.css** to component.scss.
- Add rules using SASS variables:

```
$font : "helvetica";
$brand-color : orangered;

.shop{
    font-family: $font;
    color: $brand-color;
    font-size: 1.25rem;
    text-align: center;
}
```

- Update the component to refer to the new file name:

```
styleUrls: ['./app.component.scss']
```

- A styled heading should appear at **http://localhost:4200/**

Components

- Components are **the fundamental building block** of Angular.
- They define custom-tags, which encapsulate logic, tags, style, and templates.
- **Angular 1.5** had components, and Angular 1.4 referred to them as element-directives.
- We use an **annotation** called a **Decorator** pattern, which associates metadata with an ES6 class.
- The class definition must **immediately follow** the Component.

Class properties

- The Angular-CLI will transpile our **Typescript** code back to standard Javascript.
- Here we defined a typed variable and assign a value to it in the constructor.
- The **constructor** function runs when we use an instance of this component in a template.
- Edit the template and add this code.

```
export class AppComponent {
    shop:string;

    constructor() {
        this.shop = "Southwold Organics";
    }
}
```

String interpolation : moustache syntax.

- We can refer to the class property using **string interpolation/moustache syntax** in the template.

```
<section class="shop">
<h1>{{ shop }}</h1>
</section>
```

Class methods

- Define an address object in Typescript.

```
addr:{ street:string, postcode:string };
```

- Add data to the address in the constructor.

```
this.addr = {
street : "14 Dolphin Street, Southwold",
postcode : "IP18 4HZ"
}
```

- Define a getter method in the class, which returns the string and postcode as a string.
- Note the function has a Typescript return type.

```
getAddr() : string {
    return this.addr.street + " "
        + this.addr.postcode
}
```

- Use the method in the template:

```
    <section class="shop">
        <h1>{{ shop }}</h1>
        <h2>{{ getAddr() }}</h2>
    </section>
```

- Style this h2 in the SASS file.

```
    .shop h2{
        color:darkslategrey;
        font-size: 1rem;
    }
```

Iteration/looping using *ngFor

- The Angular **ngFor** structural directive can be used to loop over an array of data.

- ngFor will generate markup in the template for each item in the array.
- Define a Typescript array of strings in the template.

```
fruit:string[];
```

- Add data to the array in the constructor.

```
this.fruit = [ "Apples","Pears" ];
```

- **Iterate** over this using ngFor in the template:

```
<section>
    <span *ngFor="let f of fruit">{{ f }}</span>
</section>
```

- Style the spans to display in a row:

```
span{
    display: inline-block;
    padding:0.5rem;
    border:1px solid lightgrey;
    margin:0.2rem;
    font-size: 1rem;
}
```

An array of objects

- Add a **price** for each fruit.
- Define **an array of objects** in Typescript.

```
fruit:{ type: string, price: number }[]

this.fruit = [
{ type: "Apples", price: 1.45 } .... ]
```

- We can change the template to use a **CSS FlexBox** (with a class of "produce") containing items (with a class of "fruit".)

```
<section class="produce">
```

```
<section class="fruit" *ngFor="let f of fruit">
    <p>{{ f.type }}</p>
    <p>{{ f.price }}</p>
</section>
```

- This requires new CSS.

```
.produce{
    display: flex;
    justify-content: center;
    flex-direction: row;
    flex-wrap: wrap;

    margin-bottom: 2rem;
}

.fruit{
    padding: 1rem;
    border:1px solid lightgrey;
    margin:0.25rem;
    text-align: center;
    font-size: 1rem;
    width: 10%;
    cursor: pointer;
}

.fruit:hover{
    background-color: beige;
}
```

Currency pipe

- PIPEs are useful where we want to transform data in the view, but not in the underlying logic/model.
- We can use an Angular **pipe** to format the price. *Note: the syntax of the currency pipe changed in Angular 5.*

```
<p>{{ f.price | currency:"GBP":"£" }}</p>
```

Angular event handlers: click to buy fruit.

- We want to be able to add fruit to a basket by clicking on them. We need to define an Angular event handler.
- Define a **click event** in the template which will call a method in the component class.

```
<section
    class="fruit"
    *ngFor="let f of fruit"
    (click)="buyFruit(f)">
```

- Add a method in the component.

```
buyFruit( f ) : void {
    console.log(f);
}
```

- An object should be logged to the console when a fruit is selected:

```
// Browser console:
{type: "Apples", price: 1.45}
```

- Define and initialise a basket array. The selected fruit will be pushed into this array.

```
basket = [];
```

- Push the selected fruit into the basket in buyFruit()

```
 this.basket.push( f );
```

- Optionally, add debugging code:

```
 console.log( JSON.stringify( this.basket ))
```

- Create code in the template to iterate over the items in the basket array.
- Note how the markup follows the same pattern as the FlexBox of fruit items created earlier.

```
<section class="produce">
```

```
    <section class="fruit"
    *ngFor="let f of basket">
        <p>{{ f.type }}</p>
        <p>{{ f.price  }}</p>
    </section>
</section>
```

Basket total

- Calculate and display the total cost of the basket.
- Define and initialise the basket total:

```
total:number=0;
```

- Add each selected fruit to the total in buyFruit()

```
this.total += f.price;
```

- Display the total in the template, *outside* of the Flexbox.
- Use a pipe to format the price.

```
<p>{{ total | currency:"GBP":"£" }}</p>
```

- Add an *ngIf directive and a truthy expression to only display the total once at least one item has been selected:

```
<p *ngIf="total" ..>
```

ngClass for conditional styling

- Extend each object to carry **instock** and **discount** properties.
- We can use the **ngClass** directive to conditionally apply a CSS class based on the state of these properties.

```
{ type: "Apples", price: 1.45, instock:true, discount:0 }
```

- If the fruit array has been typed in Typescript, its definition will need to be updated.

```
fruit:{ type: string, price:
number,instock:boolean,discount:number }[]
```

- In the template apply the outstock CSS class to each panel if that item is not in stock.

```
*ngFor="let f of fruit"
[ngClass]="{ 'outstock' : !f.instock }"
```

- Style the price field based on the discount value.
- Change the price expression to deduct the discount value.

```
<p [ngClass]="{ 'discount' : f.discount }">
{{ f.price - f.discount | currency:"GBP":"£" }}</p>
```

- Define CSS classes to highlight this change.

```
.outstock{
    opacity: 0.4;
    cursor:default;
}

.discount{
    color:forestgreen;
}
```

- Currently the out of stock items are still selectable.
- Add a truthy expression to the click handler.

```
(click)="f.instock && buyFruit(f)"
```

- *A limitation of this project is that everything is held in one large component. We should think about architecting/composing components together.*