Angular 6 Training Course

Exercise K-LifeCycle

- We will define hooks/methods that run at specific moments in the **life-cycle** of a component.
- Rebuild the existing project.

```
npm install
ng serve --open
```

- We will write lifecycle methods in tincan/tincan.component.ts.
- These methods will implement interfaces defined within Angular.
- For example, an **ngOnInit** method will implement an **OnInit** interface.
- These interfaces are imported and listed after the implements keyword in the class definition.

```
import {
    Component,
    Input,
    OnInit,
    OnDestroy,
    OnChanges,
    DoCheck,
    SimpleChanges
} from '@angular/core';

export class TincanComponent
implements OnInit,OnDestroy, OnChanges,DoCheck { .. }
```

Constructor

- The template app.component.html contains an instance of the TinCan component.
- We pass in two inputs: **price** and **product**.

```
<tincan [product]="product" [price]="price"
*ngIf="state"></tincan>
```

- If we attempt to log product and price in the constructor of **tincan.component.ts** it displays **undefined**.

```
    constructor() {
```

```
    console.log( "constructor" );
    console.log( this.product, this.price );
    }
```

- Input bindings for a component are not yet defined/established in the constructor.
- However, we can use **dependency injection (DI)** in the constructor to call a service.
- A **shop service** is defined with a **getName method**.
- We can inject an instance of this into the constructor.

```
    constructor( shop:ShopService ) { .. }
```

- We can set the name property using this method.

```
    this.name = shop.getName();
```

OnInit

- To see the Input bindings, console.log them in **ngOnInit**..

```
    ngOnInit() {
    console.log("ngOnInit", this.product, this.price );
```

- Both DI and Input bindings will be working at this point.
- This method is a good choice to request data from a service.

OnDestroy

- In the main component template, we have defined an **ngIf** directive.
- We have defined a boolean variable called state.
- If this is set to false, a component instance will be removed from the DOM and an **ngOnDestroy** method triggered.
- If it is set to true, a component instance will be recreated in the DOM and an **ngOnInit** method will be triggered.

```
<tincan .. *ngIf="state"></tincan>
```

- The remove and create buttons already have click handlers set up.

```
    <p (click)="remove()">Remove</p>
    <p (click)="create()">Create</p>
```

- Add logic to set this boolean variable.

```
    remove() {
    this.state = false;
```

```
}

create() {
this.state = true;
}
```

- Add debugging to the ngOnDestroy method.

```
ngOnDestroy() {
console.log("ngOnDestroy" , this);
```

- Review this in the Chrome web tools Elements tab. You can see the TinCan component being removed and added to the DOM.

Removing Observable subscribers in ngOnDestroy.

- Create an Observable by calling createSequence in the constructor.

```
this.createSequence();
```

- Run this. It displays the number sequence on the page and in the console.
- Click the remove button
- Note that the Observable subscriber code continues to run in the console.
- To ensure that the Observable is correctly removed, add code in ngOnDestroy to unsubscribe from the Observable.

```
ngOnDestroy() {
    console.log("ngOnDestroy" , this);
    this.subscription.unsubscribe();
}
```

ngOnChanges

- Add this life-cycle hook which is invoked when there are changes to the **INPUTS** passed into a component.

```
ngOnChanges( changes: SimpleChanges ) {
console.log( JSON.stringify( changes ));
}
```

- Add an increasePrice method to the main component. This will change the **price input**.

```
increasePrice() {
this.price += 0.10;
}
```

- Click price and note the changes object logged to the console.

```
{"price":
{"previousValue":0.45,"currentValue":0.55,"firstChange":false}}
```

ngOnChanges and complex arrays/objects.

- Add similar code to the buyStock method which reduces the stock property of the product object down to a minimum of zero.

```
buyStock() {
    this.product.stock = Math.max( this.product.stock-1 , 0 );
}
```

- Note this does reduce the displayed quantity, but does not trigger a call to **ngOnChanges**.
- Changes to the properties of complex arrays and objects do not trigger this event.
- Changes to values of primitive strings and numbers do.
- There are alternative solutions to make this work.
- This code makes a true independent copy of the object before changing it. This does trigger an ngOnChange event.

```
    buyStock() {
    this.product = Object.assign( {} , this.product );
    this.product.stock = Math.max( this.product.stock-1 , 0 );
    }
```

ngDoCheck

- Alternatively, we can comment out the buyStock method and implement logic in the ngDoCheck method which will pick up changes to properties of objects/arrays.

```
    buyStock() {
    // this.product = Object.assign( {} , this.product );
    this.product.stock = Math.max( this.product.stock-1 , 0 );
    }
    ngDoCheck() {
        console.log("ngDoCheck", this.price, this.product );
    }
```

- Note ngDoCheck may be invoked by many events/interactions in your application and may carry a big performance cost.