

## Angular 6 Training Course

### Exercise C-compose

- Angular components are **composable**.
- Angular projects consist of a **hierarchy** of related components.
- This exercise **refactors** example **b-shop** into separate components.
- Angular **Inputs** will pass data **in** to components.
- Angular **Outputs** will emit events **out** of components.

#### Setup

- Delete the node\_modules folder in **b-shop**.
- Duplicate and rename the folder as **c-compose**.

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

- Check that the page still functions correctly.

#### Create a component for each item of fruit.

- The Angular CLI can **generate** new components.
- It updates **app/app.module.ts** to reflect those changes.
- Use the --dry-run option to find out what files would be created before creating them.

```
ng generate component fruit --dry-run
```

- Run the command again without --dry-run to make the change.

```
ng generate component fruit
```

- Iterate over the new fruit component in **app.component.html**

```
<app-fruit *ngFor="let f of fruit"></app-fruit>
```

- This works but passes no information down to the fruit component.

#### Inputs

- We pass data in to the fruit component using an **Input**.
- Import the input class.

```
import { Input } from '@angular/core';
```

- Define the input decorator inside the class before the constructor.

```
@Input() fruit;
```

- We can then pass a fruit object into each instance of this component.
- [fruit] defines the Input and "f" refers to the temporary variable created by the ngFor iterator.

```
<app-fruit  
  *ngFor="let f of fruit"  
  [fruit]="f">  
</app-fruit>
```

- Add debugging into the fruit component class to prove that the fruit object is being passed down.

```
ngOnInit() { console.log(this.fruit);}
```

- The fruit object will be logged to the browser console.

```
{type: "Pears", price: 1.85, instock: true, discount: 0.4}
```

- Note, the Angular component **LifeCycle** means that the input does not come into existence until the OnInit method.

### **Define the fruit template**

- Edit **app/fruit/fruit.component.html** to define the fruit component template.

```
<section class="fruit">  
  <p>{{ fruit.type }}</p>  
  <p>{{ fruit.price }}</p>  
</section>
```

- Adjust the price to take account of the **discount**.
- Format the price with a built-in **Angular Pipe**.

```
<p>{{ fruit.price - fruit.discount | currency:"GBP":"£" }}</p>
```

- Conditionally style the section using an ngClass directive.

```
<section class="fruit"  
  [ngClass]="{ 'outstock' : !fruit.instock }">
```

### **Emit Events using an Output Decorator**

- When the user clicks on a fruit, we want the fruit component to emit an event.
- The main component template will listen for this event and runs its buyFruit method to push fruit into an array.

- Import and define an Output before the constructor.

```
import {EventEmitter,Output} from '@angular/core';
export class AppComponent { ..
  @Output() select = new EventEmitter();
  constructor() { ..
```

- Listen for this event in the fruit component template.

```
<section class="fruit" (click)="select.emit()">
```

- Listen for the select event in the main template

```
<app-fruit [fruit]="f" (select)="buyFruit(f)" >
```

- Add additional logic in the fruit component template to prevent out of stock items being bought.

```
(click)="fruit.instock && select.emit()"
```

### ***Create a basket in the main template.***

- Edit the main template to iterates over the basket array.

```
<section class="flex">
  <app-fruit
    *ngFor="let f of basket"
    [fruit]="f">
  </app-fruit>
</section>
```

- We do not want to call buyFruit when the user clicks on a basket item.
- We can listen for the same select event and then call removeItem().

```
(select)="removeItem(f)"
```

### ***Total***

- We can add a total and empty button to the main template.

```
<h2>{{ getTotal() | currency:"GBP":"£" }}</h2>
```

```
<p *ngIf="basket.length" class="empty"
(click)="empty()">Empty</p>
```

### ***Custom Type***

- We can define a Typescript custom type for each stock item.
- Create a new file **types/item.type.ts**

```
export interface Item{  
    type: string,  
    price: number,  
    instock:boolean,  
    discount:number  
}
```

- Import the type into the main class.

```
import { Item } from './types/item.type';  
fruit:Item[];
```