# Component Interactions

1. Component hierarchies
2. Component inputs

- Component outputs

```
Demo app:   AngularDev/Demos/05-ComponentInteractions/DemoApp
To install: npm install
To run:     ng serve
```

# Section 1: Component Hierarchies

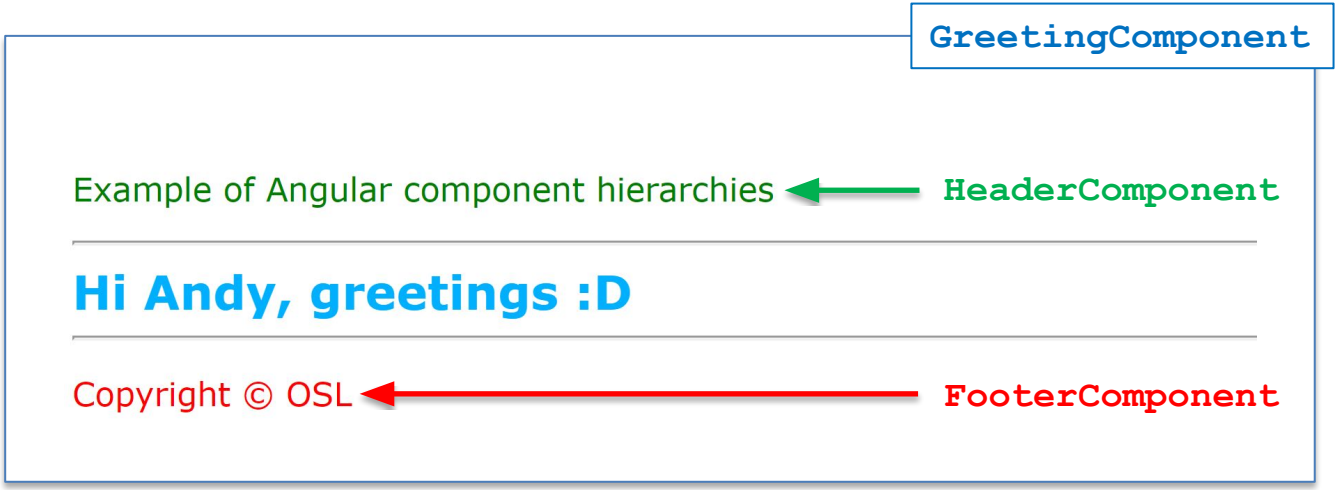
- Overview
- Component hierarchy example
- Component code organization
- Low-level components
- High-level component
- Bundling components

# Overview

- An Angular application typically has many components
  - Each component renders a portion of UI real estate
- If a component is starting to get too complex…
  - You can split it into lower-level components
  - Just like you split complex functions into smaller ones
- This creates a more modular application:
  - Components are simpler, easier to test, possibly reusable

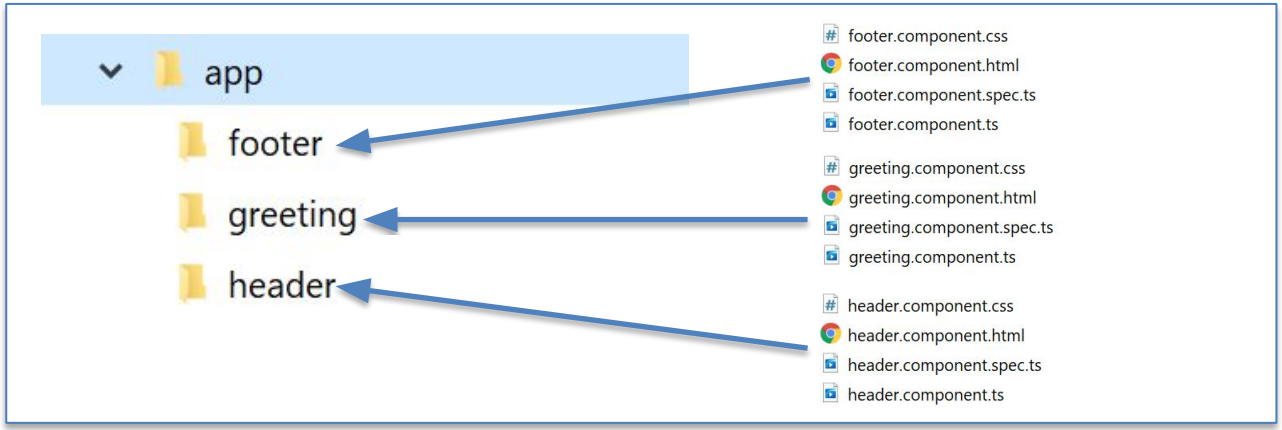
# Component Hierarchy Example

- Let's see an example of a component hierarchy
  - In the demo app, click the **Component hierarchy** link



# Component Code Organization

- It's common to put each component into a separate folder in your application
  - A folder contains all the code files for a component



# Low-Level Components

- Here are the header and footer components
  - Also see the HTML and CSS files for each component

```
@Component({
  selector: 'app-header',
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.css']
})
export class HeaderComponent {}
```

header.component.ts

```
@Component({
  selector: 'app-footer',
  templateUrl: './footer.component.html',
  styleUrls: ['./footer.component.css']
})
export class FooterComponent {}
```

footer.component.ts

# High-Level Component (1 of 2)

- Here's the high-level greeting component code:

```
@Component({
  selector: 'app-greeting',
  templateUrl: './greeting.component.html',
  styleUrls: ['./greeting.component.css']
})
export class GreetingComponent {
  name = 'Andy'
}
```

**greeting.component.ts**

# High-Level Component (2 of 2)

- Here's the high-level greeting component HTML:

```
<app-header></app-header>

<div class="greetingContent">
  Hi {{name}}, greetings :D
</div>

<app-footer></app-footer>
```

greeting.component.html

- Note:
  - `<app-header>` tag instantiates `HeaderComponent`
  - `<app-footer>` tag instantiates `FooterComponent`

# Bundling Components

- Remember to bundle your components into a module

```
import { GreetingComponent } from './greeting/greeting.component';
import { HeaderComponent } from './header/header.component';
import { FooterComponent } from './footer/footer.component';
…

@NgModule({
  declarations: [GreetingComponent, HeaderComponent, FooterComponent, … ],
  …
})
export class AppModule { }
```

app.module.ts

- In a small app, put all components in the root module
  - In a large app, you can define multiple *feature modules*

# Section 2: Component Inputs

- Overview
- Component inputs example
- Specifying an input property
- Setting an input property

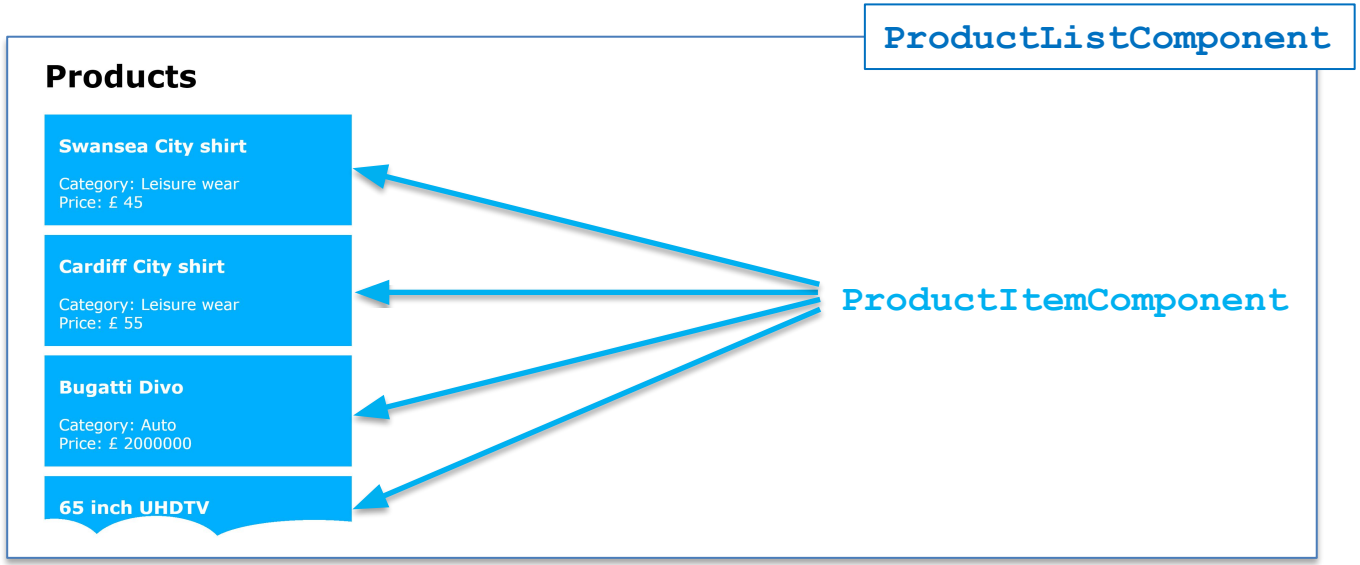
# Overview

- Imagine you have a hierarchy of components…
  - A high-level component gets data from somewhere
  - The high-level component wants to pass the data into a lower-level component, to be displayed etc.

- Angular allows a high-level component to pass a value into a low-level component property
  - In low-level component, decorate property with `@Input`
  - In high-level component, assign a value to property

# Component Inputs Example

- Let's see an example of component inputs
  - In the demo app, click the **Component inputs** link



# Specifying an Input Property

- To specify that a component's property will be input from another component:
  - Decorate the property with `@Input()`

```
import { Component, Input } from '@angular/core';
import { Product } from '../product'

@Component({
  selector: 'app-product-item',
  templateUrl: './product-item.component.html',
  styleUrls: ['./product-item.component.css']
})
export class ProductItemComponent {
  @Input()
  product!: Product;
}
```

product-item.component.ts

# Setting an Input Property

- To set the value for a component's input property:
  - Use `[]` to specify the property name
  - Supply a suitable input value

```
<h1>Products</h1>

<div>
    <div *ngFor="let p of products">
        <app-product-item [product]="p"></app-product-item>
    </div>
</div>
```

**product-list.component.html**

# Summary

- Component hierarchies
- Component inputs

# Annex:  Component Outputs

- Overview
- Component outputs example
- Defining an event interface
- Emitting events
- Aside: Lifecycle hooks
- Handling events

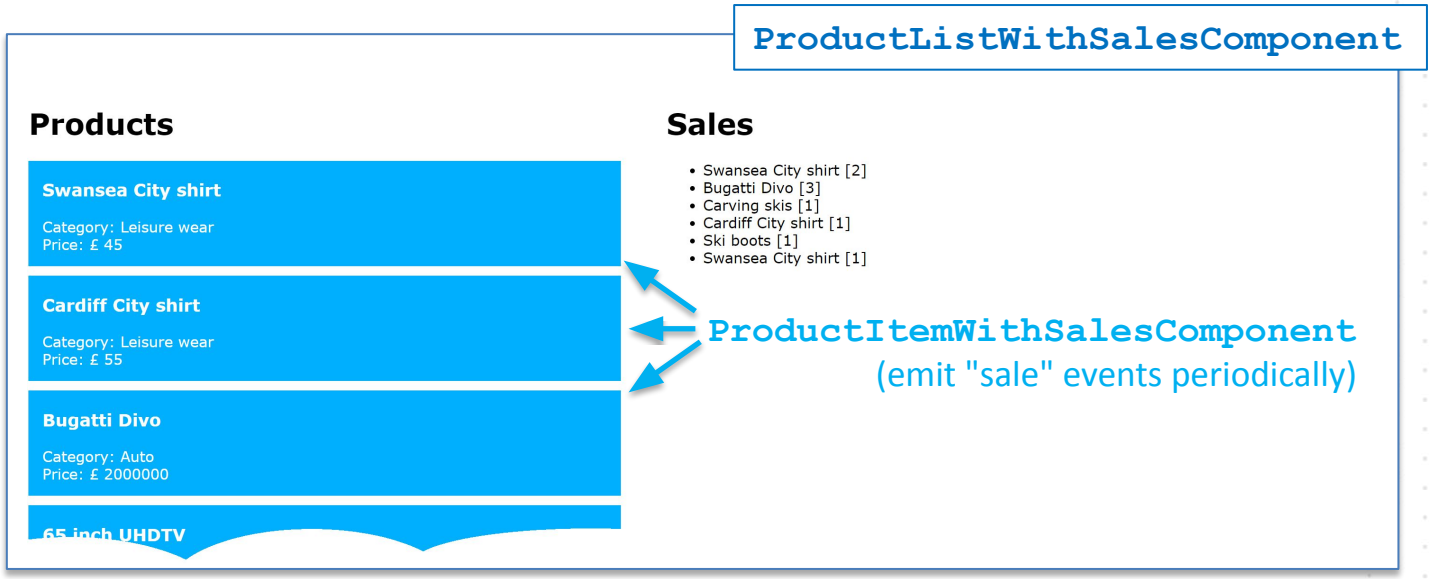
# Overview

- Imagine you have a hierarchy of components...
  - A low-level component modifies data and wants to notify any "interested listeners" about these changes
  - A high-level component is an interested listener, i.e. it wants to know about these changes

- Angular allows components to emit events
  - In low-level component, emit event when data changes
  - In high-level component, define event-handler for event

# Component Outputs Example

- Let's see an example of component outputs
  - In the demo app, click the **Component outputs** link



# Defining an Event Interface

- The first step is to define an event interface
  - Specifies info the low-level component wants to pass up to the high-level component

- E.g. `ProductItemWithSalesComponent` emits a "sale" event periodically
  - We represent the "sale" event via the `ISale` interface

```
export interface ISale {
  productDescription: string;
  quantity: number;
};
```

**product-item-with-sales.component.ts**

# Emitting Events

- To emit an event from a component:
  - Define `EventEmitter` property as an `@Output()`
  - Call `emit()` to emit an event, and pass event data

```
@Component(…)
export class ProductItemWithSalesComponent implements OnInit {

  @Input()
  product!: Product;

  @Output()
  sale: EventEmitter<ISale> = new EventEmitter();

  ngOnInit() {
      setInterval(() => {
          let eventData: ISale = {…}
          this.sale.emit(eventData);
      }, 1000 + (5000 * Math.random()));
  }
}
```

product-item-with-sales.component.ts

# Aside: Lifecycle Hooks

- `ProductItemWithSalesComponent` implements the `OnInit` interface
  - This is an Angular lifecycle hook

- Angular calls the component's `ngOnInit()` method
  - After object has been constructed and inputs applied

```
import {OnInit, … } from '@angular/core';

@Component(…)
export class ProductItemWithSalesComponent implements OnInit {

  ngOnInit() { /* Do any initialization that uses @Input() properties */ }
}
```

# Handling Events

- To handle events from low-level component property:
  - Use () to specify property name that might emit events
  - Define a suitable event handler function

```
<div *ngFor="let p of products">
    <app-product-item-with-sales [product]="p" (sale)="onSale($event)">
    </app-product-item-with-sales>
</div>
```

product-list-with-sales.component.html

```
@Component(…)
export class ProductListWithSalesComponent {
  products: Array<Product> = [];
  sales: Array<string> = [];
  …
  onSale(event: ISale) {
   let msg:string = event.productDescription + ' [' + event.quantity + ']';
   this.sales.push(msg);
  }
}
```

product-list-with-sales.component.ts