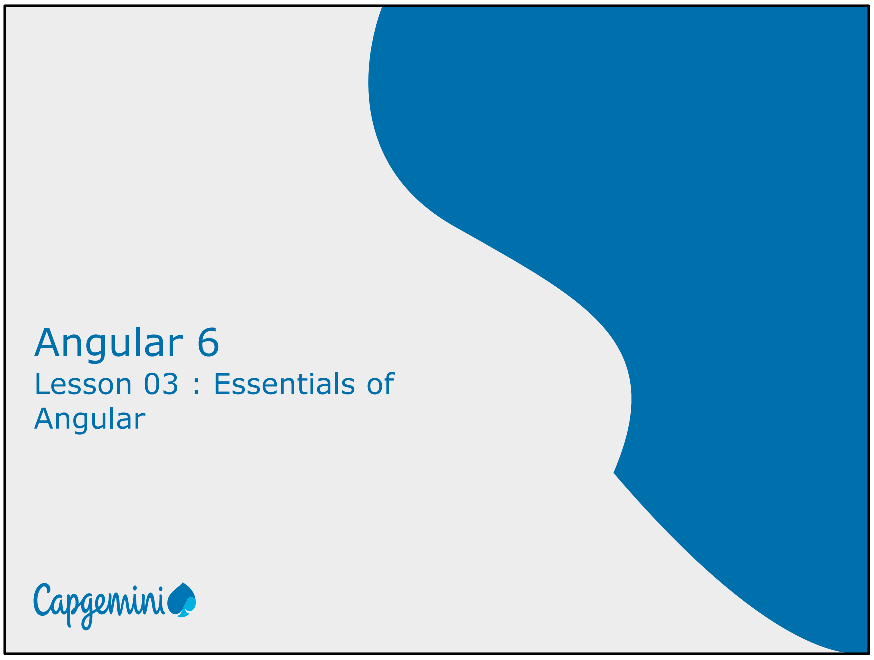


Angular 6

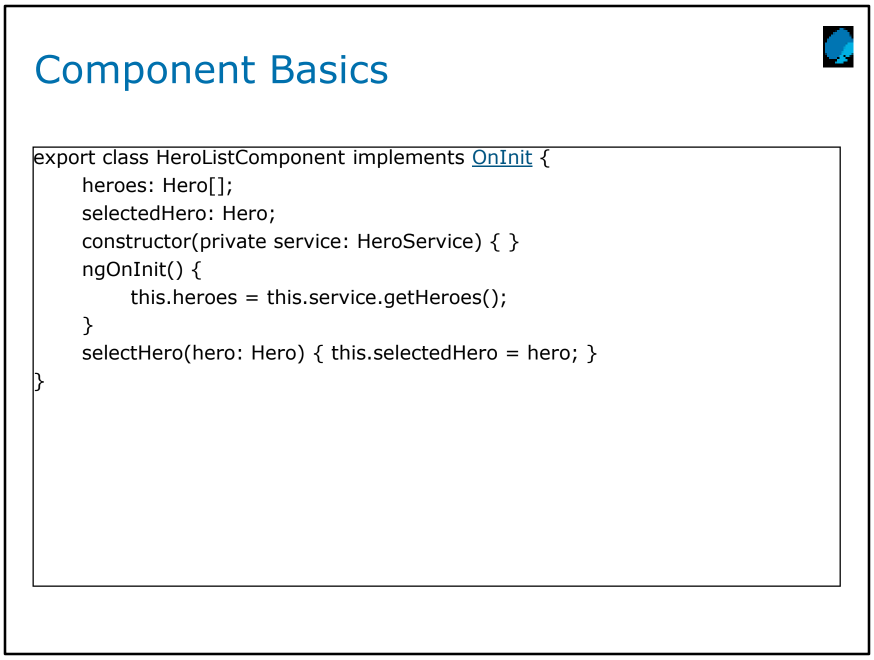
Lesson 03 : Essentials of Angular

Lesson Objectives

- Component Basics
- Setting up the templates
- Creating Components using CLI
- Nesting Components
- Data Binding - Property & Event Binding, String Interpolation, Style binding
- Two-way data binding
- Input Properties, Output Properties, Passing Event Data
- Case Study

Component Basics

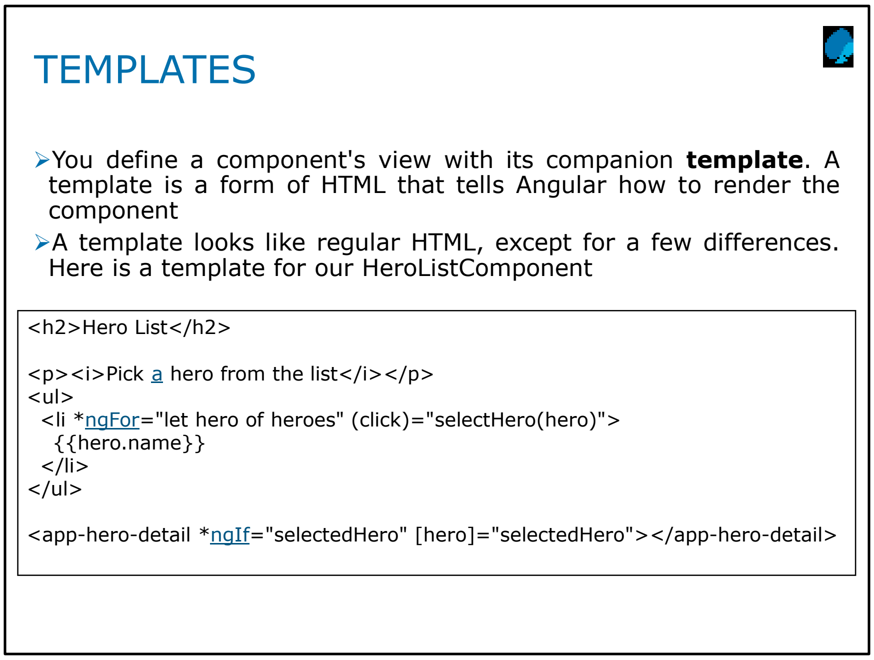
- A component controls a patch of screen called a view
- You define a component's application logic—what it does to support the view—inside a class.
- The class interacts with the view through an API of properties and methods.
- For example, this HeroListComponent has a heroes property that returns an array of heroes that it acquires from a service. HeroListComponent also has a selectHero() method that sets a selectedHero property when the user clicks to choose a hero from that list.



Component Basics

```
export class HeroListComponent implements OnInit {
    heroes: Hero[];
    selectedHero: Hero;
    constructor(private service: HeroService) { }
    ngOnInit() {
        this.heroes = this.service.getHeroes();
    }
    selectHero(hero: Hero) { this.selectedHero = hero; }
}
```

Angular creates, updates, and destroys components as the user moves through the application. Your app can take action at each moment in this lifecycle through optional **lifecycle hooks**, like ngOnInit() declared above.

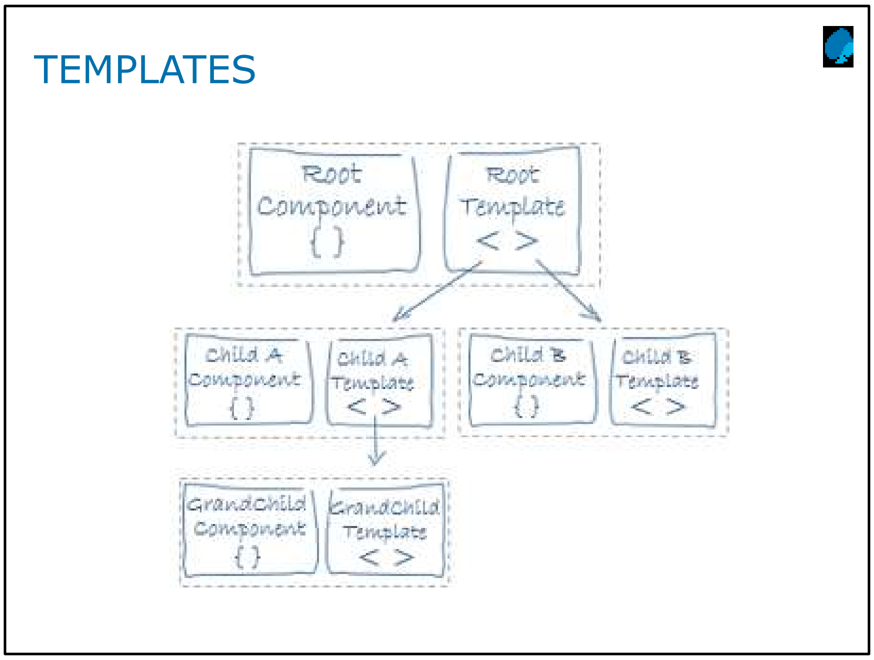


TEMPLATES

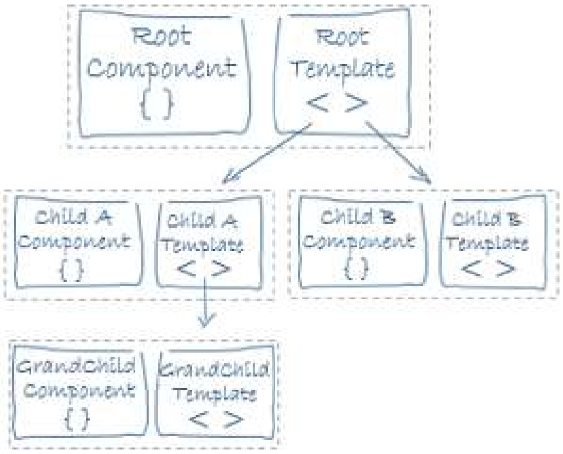
- You define a component's view with its companion **template**. A template is a form of HTML that tells Angular how to render the component
- A template looks like regular HTML, except for a few differences. Here is a template for our HeroListComponent

```
<h2>Hero List</h2>

<p><i>Pick a hero from the list</i></p>
<ul>
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
    {{hero.name}}
  </li>
</ul>

<app-hero-detail *ngIf="selectedHero" [hero]="selectedHero"></app-hero-detail>
```

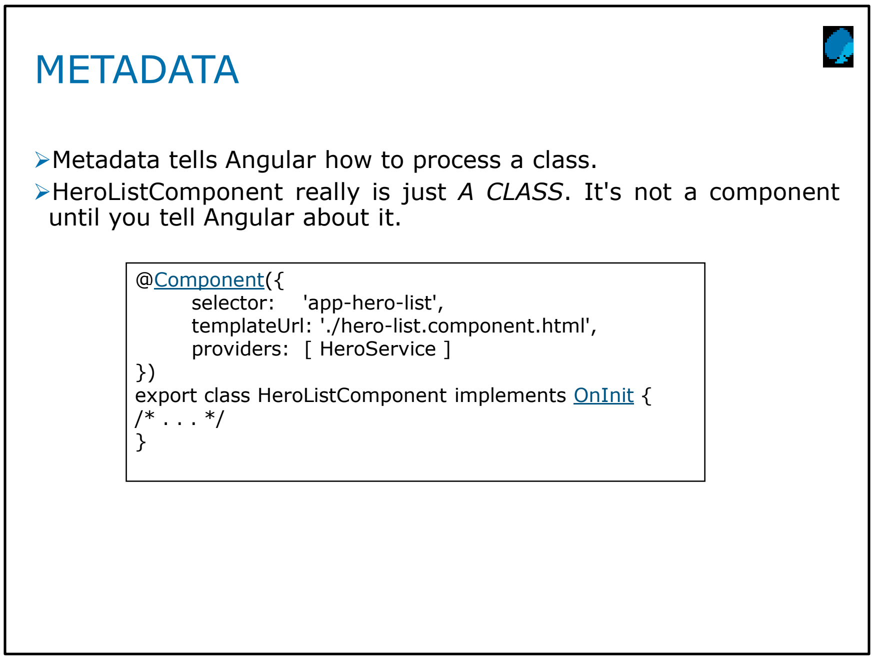
Although this template uses typical HTML elements like <h2> and <p>, it also has some differences. Code like *ngFor, {{hero.name}}, (click), [hero], and <hero-detail> uses Angular's template syntax.
In the last line of the template, the <hero-detail> tag is a custom element that represents a new component, HeroDetailComponent.
The HeroDetailComponent is a different component than the HeroListComponent you've been reviewing. The HeroDetailComponent (code not shown) presents facts about a particular hero, the hero that the user selects from the list presented by the HeroListComponent. The HeroDetailComponent is a **child** of the HeroListComponent.



TEMPLATES



Notice how <hero-detail> rests comfortably among native HTML elements. Custom components mix seamlessly with native HTML in the same layouts

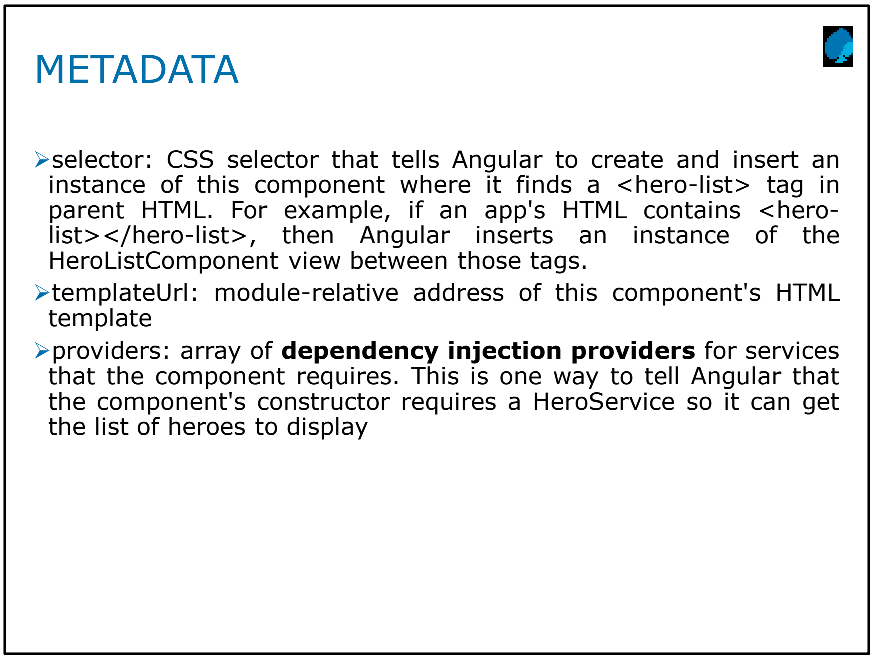


METADATA

- ➢Metadata tells Angular how to process a class.
- ➢HeroListComponent really is just *A CLASS*. It's not a component until you tell Angular about it.

```
@Component({
        selector:    'app-hero-list',
        templateUrl: './hero-list.component.html',
        providers:  [ HeroService ]
})
export class HeroListComponent implements OnInit {
/* . . . */
}
```

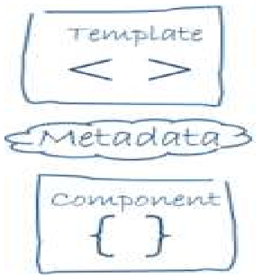
Here is the @Component decorator, which identifies the class immediately below it as a component class. The @Component decorator takes a required configuration object with the information Angular needs to create and present the component and its view.



METADATA

- selector: CSS selector that tells Angular to create and insert an instance of this component where it finds a <hero-list> tag in parent HTML. For example, if an app's HTML contains <hero-list></hero-list>, then Angular inserts an instance of the HeroListComponent view between those tags.
- templateUrl: module-relative address of this component's HTML template
- providers: array of **dependency injection providers** for services that the component requires. This is one way to tell Angular that the component's constructor requires a HeroService so it can get the list of heroes to display

Here is the @Component decorator, which identifies the class immediately below it as a component class. The @Component decorator takes a required configuration object with the information Angular needs to create and present the component and its view.

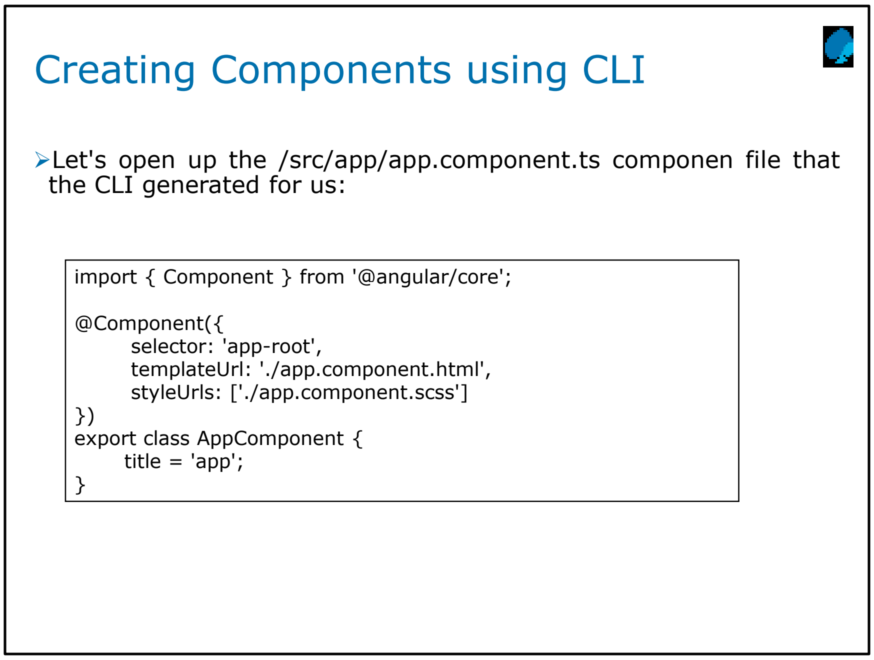


```
@Component({
         selector:    'app-hero-list',
         templateUrl: './hero-list.component.html',
         providers:  [ HeroService ]
})
export class HeroListComponent implements OnInit {
       heroes: Hero[];
       selectedHero: Hero;
       constructor(private service: HeroService) { }
       ngOnInit() {
              this.heroes = this.service.getHeroes();
       }
       selectHero(hero: Hero) { this.selectedHero = hero; }
}
```

Creating Components using CLI

- Angular components are the basic building blocks of your app. Each component defines:
- Any necessary imports needed by the component
- A component decorator, which includes properties that allow you to define the template, CSS styling, animations, etc..
- A class, which is where your component logic is stored.
- Angular components reside within the /src/app folder:

```
> src
  > app
    app.component.ts      // A component file
    app.component.html  // A component template file
    app.component.css    // A component CSS file
    > about               // Additional component folder
    > contact             // Additional component folder
```

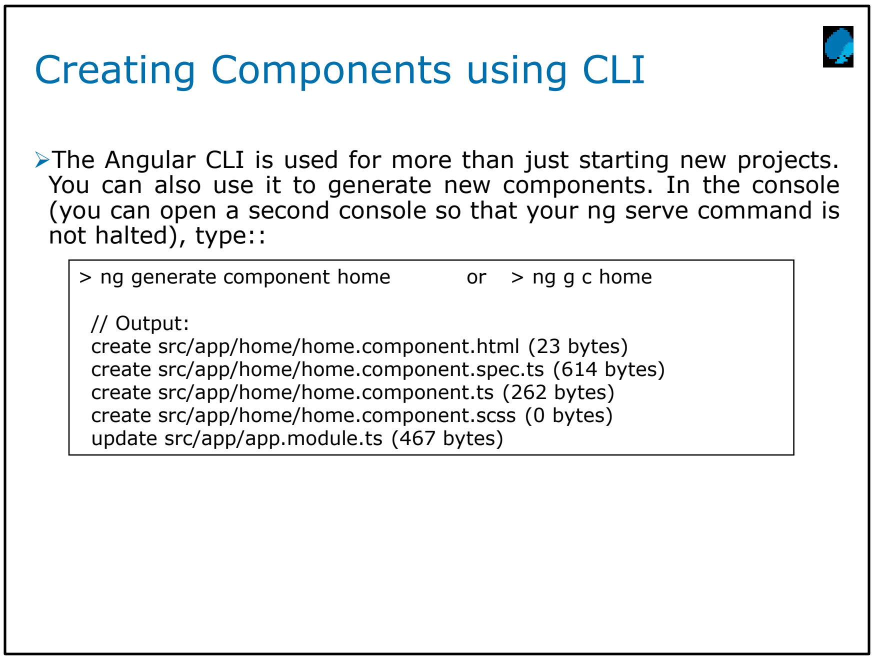


Creating Components using CLI

- Let's open up the /src/app/app.component.ts componen file that the CLI generated for us:

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

@Component({
        selector: 'app-root',
        templateUrl: './app.component.html',
        styleUrls: ['./app.component.scss']
})
export class AppComponent {
       title = 'app';
}
```

As mentioned previously, at the top we have our imports, the component decorator in the middle (which defines the selector, template and style location), and the component class.
Notice the selector **app-root**? If you open /src/index.html you will see a custom HTML tag noted as <app-root></app-root>.
This is where that component will load! If you have ng serve running in the console, you will note that in the browser at http://localhost:4200, the HTML matches the /src/app/app.component.html file.



Creating Components using CLI

- The Angular CLI is used for more than just starting new projects. You can also use it to generate new components. In the console (you can open a second console so that your ng serve command is not halted), type::

```
> ng generate component home          or    > ng g c home

  // Output:
  create src/app/home/home.component.html (23 bytes)
  create src/app/home/home.component.spec.ts (614 bytes)
  create src/app/home/home.component.ts (262 bytes)
  create src/app/home/home.component.scss (0 bytes)
  update src/app/app.module.ts (467 bytes)
```

Let’s look at the component code and then take these one at a time. Open our first TypeScript file: src/app/home/home.component.ts

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

@Component({
selector: 'app-home',
templateUrl: './home.component.html',
styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {
constructor() { }
ngOnInit() {
}
}
```

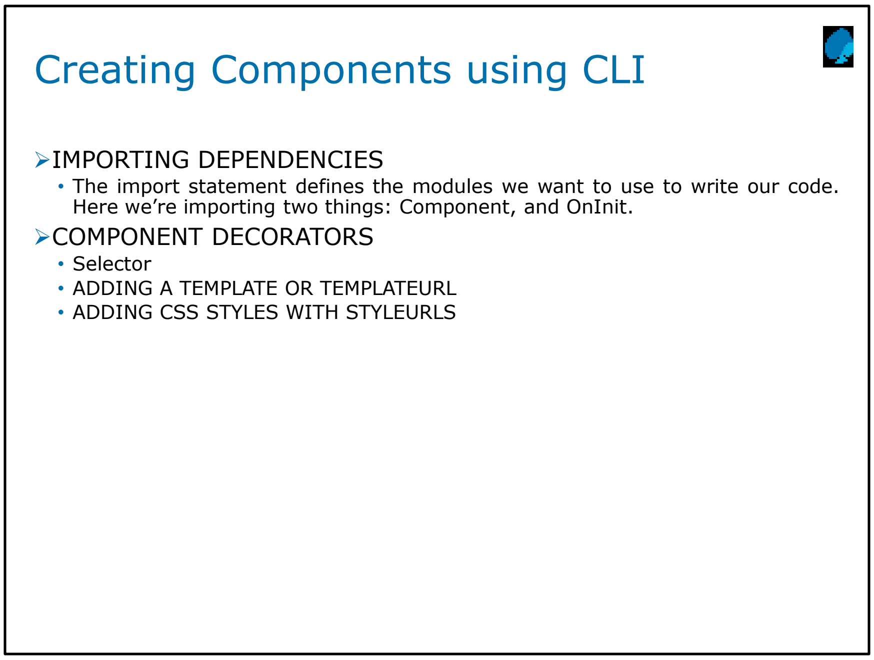
Notice that we suffix our TypeScript file with .ts instead of .js The problem is our browser doesn’t know how to interpret TypeScript files. To solve this gap, the ng serve command live-compiles our .ts to a .js file automatically

Creating Components using CLI

- Let’s look at the component code and then take these one at a time. Open our first TypeScript file:

  src/app/home/home.component.ts

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

@Component({
      selector: 'app-home',
      templateUrl: './home.component.html',
      styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {
      constructor() { }
      ngOnInit() {
      }
}
```

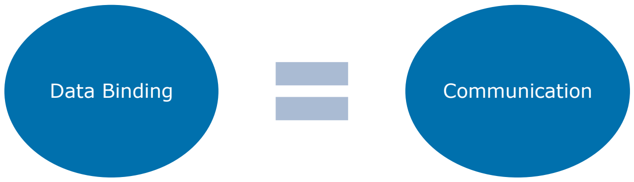


Creating Components using CLI

- IMPORTING DEPENDENCIES
  - The import statement defines the modules we want to use to write our code. Here we’re importing two things: Component, and OnInit.
- COMPONENT DECORATORS
  - Selector
  - ADDING A TEMPLATE OR TEMPLATEURL
  - ADDING CSS STYLES WITH STYLEURLS

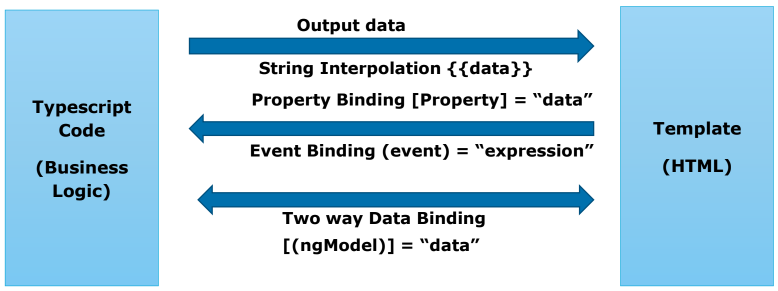
Data Binding

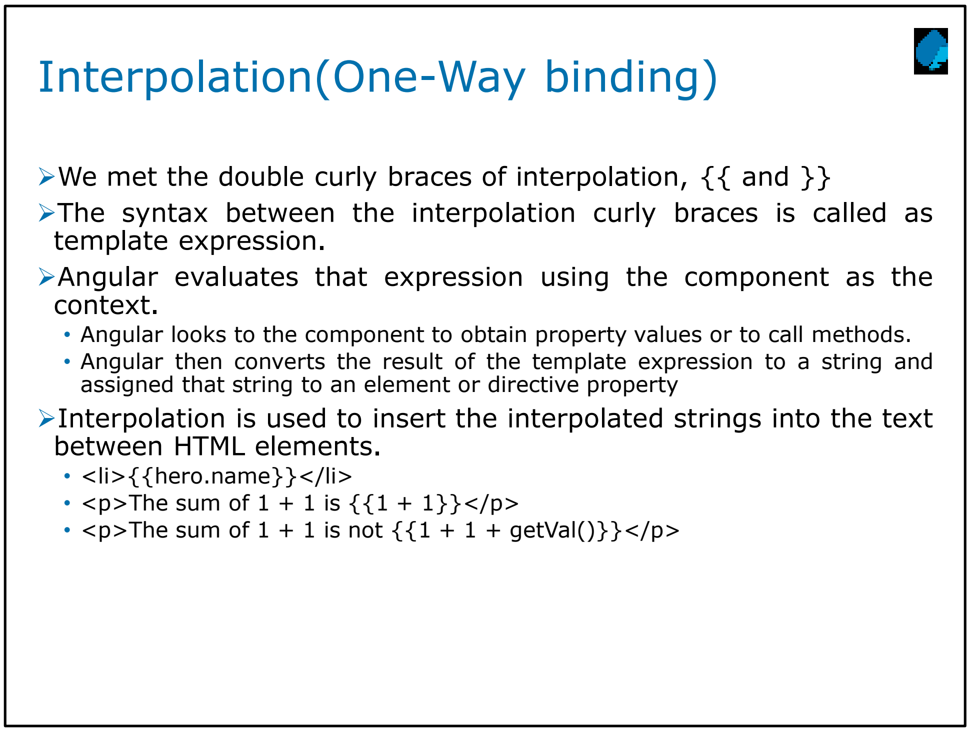
- Communication between your typescript code of your business logic and the template what the user sees is called Data Binding



Data Binding

- Communication between your typescript code of your business logic and the template what the user sees is called Data Binding


Typescript Code
(Business Logic)
Output data
String Interpolation {{data}}
Property Binding [Property] = "data"
Event Binding (event) = "expression"
Two way Data Binding
[(ngModel)] = "data"
Template
(HTML)



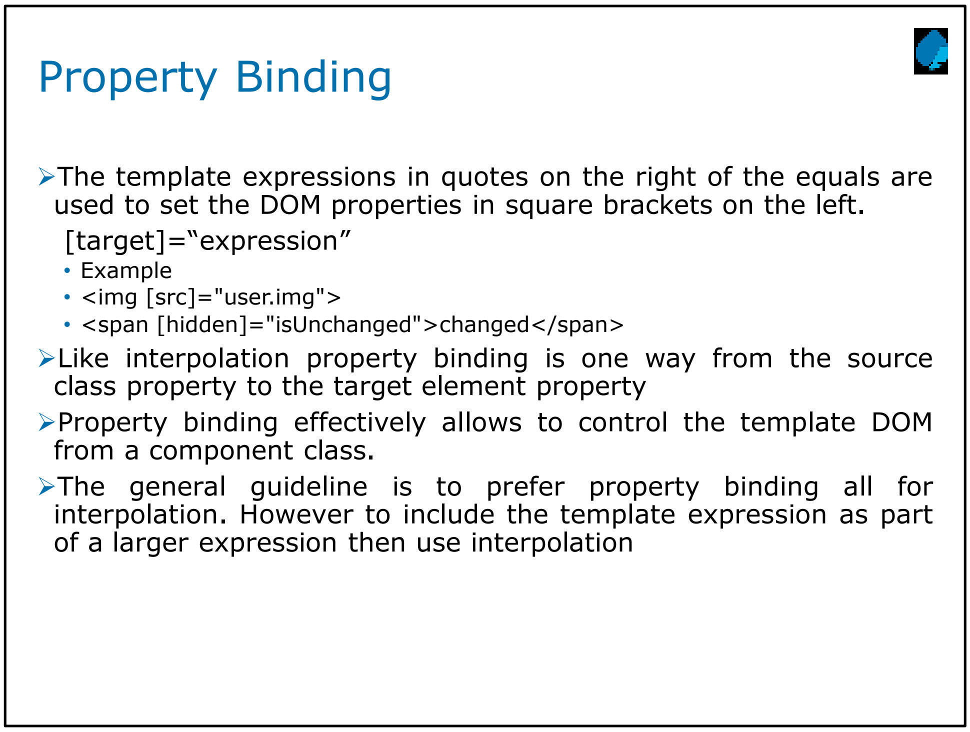
Interpolation(One-Way binding)

- We met the double curly braces of interpolation, {{ and }}
- The syntax between the interpolation curly braces is called as template expression.
- Angular evaluates that expression using the component as the context.
  - Angular looks to the component to obtain property values or to call methods.
  - Angular then converts the result of the template expression to a string and assigned that string to an element or directive property
- Interpolation is used to insert the interpolated strings into the text between HTML elements.
  - <li>{{hero.name}}</li>
  - <p>The sum of 1 + 1 is {{1 + 1}}</p>
  - <p>The sum of 1 + 1 is not {{1 + 1 + getVal()}}</p>

Interpolation with Curly Braces
Double curly braces contain *template expressions* which allow us to read primitive or object values from component properties. Template expressions are similar to JavaScript and include features such as the ternary operator, concatenation and arithmetic.
nterpolation automatically escapes any HTML, so here {{html}} displays <div>this is a div</div>, or more precisely it displays &lt;div&gt;this is a div&lt;/div&gt;.

Demo

- Demo One Way Binding



Property Binding

- The template expressions in quotes on the right of the equals are used to set the DOM properties in square brackets on the left.

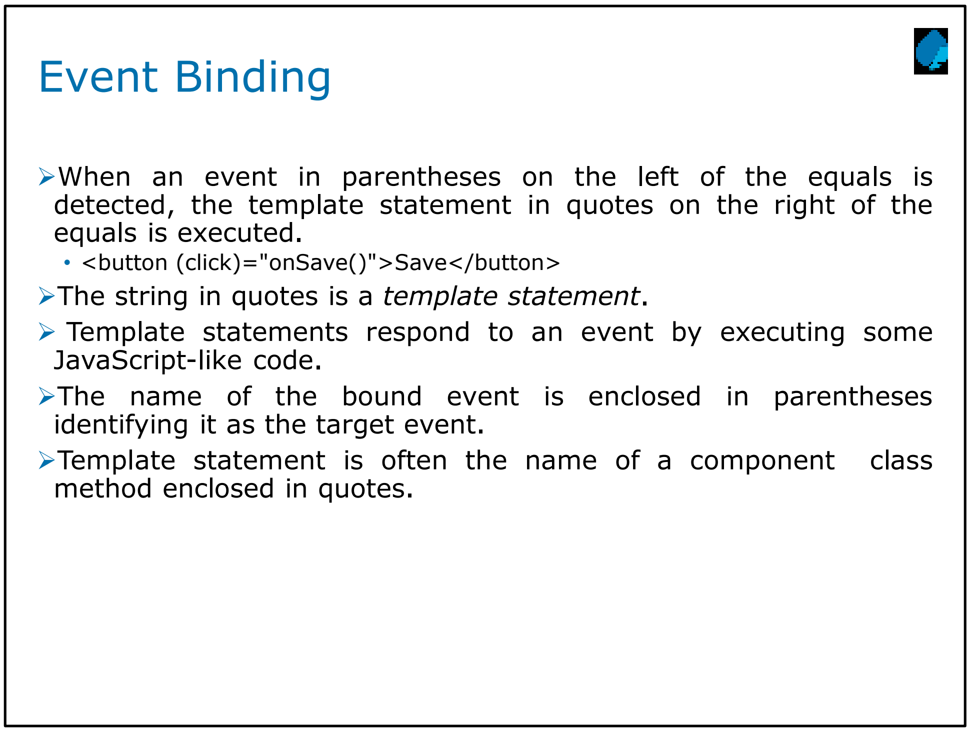
  [target]=“expression”
  - Example
  - <img [src]="user.img">
  - <span [hidden]="isUnchanged">changed</span>
- Like interpolation property binding is one way from the source class property to the target element property
- Property binding effectively allows to control the template DOM from a component class.
- The general guideline is to prefer property binding all for interpolation. However to include the template expression as part of a larger expression then use interpolation

Property Binding with Square Brackets
We specify one-way bindings to DOM properties using square brackets and template expressions.
The template expressions in quotes on the right of the equals are used to set the DOM properties in square brackets on the left.
For example, textContent is set to a person's name, buttons are disabled based on the sex property of person, and the src, alt and title properties of the img element are bound to properties of the person object.
innerHTML sets the HTML content of the span to a number of stars (&#10032; is a star as you can see in the application output below). We use [innerHTML] here rather than interpolation here because interpolation would escape the HTML
Template expressions are similar to JavaScript and include features such as the ternary operator, concatenation and boolean conditions. We can call built-in methods such as repeat as well.
First, setting an attribute on an element will not evaluate the template expression in quotes. This is just a standard attribute assignment using a string, and we can confirm this in the application output below where we see the 'Hello ' + name string and not Hello World.
However, when we use interpolation in the quotes (identified by the double curly braces), we are now binding to the DOM *property* instead, and the template expression is evaluated. This is a subtle but important difference.
Property bindings using square brackets or the bind- prefix (known as the canonical form) always interpret the string in quotes as a template expression, so there is no need to include

the curly braces in this case.

Demo

- Demo Property Binding



Event Binding

- ➢When an event in parentheses on the left of the equals is detected, the template statement in quotes on the right of the equals is executed.
  - <button (click)="onSave()">Save</button>
- ➢The string in quotes is a *template statement*.
- ➢ Template statements respond to an event by executing some JavaScript-like code.
- ➢The name of the bound event is enclosed in parentheses identifying it as the target event.
- ➢Template statement is often the name of a component class method enclosed in quotes.

The only way to know about a user action is to listen for certain events such as keystrokes, mouse movements, clicks, and touches. We declare our interest in user actions through Angular event binding.

Event binding syntax consists of a target event within parentheses on the left of an equal sign, and a quoted template statement on the right.

$event object is used to listen to an event and grab's the user event

Event Binding with Parentheses
We handle DOM events using parentheses and template statements.
When an event in parentheses on the left of the equals is detected, the template statement in quotes on the right of the equals is executed. Alternatively, we can use the *canonical* form by prefixing the event name with on-. These three bindings are equivalent to (click), (dblclick) and (contextmenu).
The string in quotes is a *template statement*. Template statements respond to an event by executing some JavaScript-like code. Here we simply call methods on the component class and pass in the $event object.

Keyboard Events:Template statements are expected to change the state of the application based on the user's input. Technically, they can contain statements that directly alter data such as "key = $event.key" but this is usually best handled by calling a method.

Checkbox---The change event is triggered when the user clicks on a checkbox

```
import { Component } from '@angular/core';
@Component({
selector: 'app-checkbox', template: ` <h1>Checkbox</h1> <p> <label
[class.selected]="cb1.checked"> <input #cb1 type="checkbox" value="one"
(change)="logCheckbox(cb1)"> One </label> <label [class.selected]="cb2.checked"> <input
#cb2 type="checkbox" value="two" (change)="logCheckbox(cb2)"> Two </label> <label
[class.selected]="cb3.checked"> <input #cb3 type="checkbox" value="three"
(change)="logCheckbox(cb3)"> Three </label> </p> <h2>Log <button
(click)="log=''">Clear</button></h2> <pre>{{log}}</pre>`, styles: ['.selected {color:
OrangeRed;}'] })
export class CheckboxComponent
{ log = ''; logCheckbox(element: HTMLInputElement): void { this.log += `Checkbox
${element.value} was ${element.checked ? '' : 'un'}checked\n`; } }
```

Text Areatextarea behaves in a similar way to the textbox

```
import { Component } from '@angular/core';
@Component({
selector: 'app-text-area', template: ` <h1>Text Area</h1> <textarea ref-textarea
[(ngModel)]="textValue" rows="4"></textarea><br/> <button
(click)="logText(textarea.value)">Update Log</button> <button
(click)="textValue=''">Clear</button> <h2>Log <button (click)="log=''">Clear</button></h2>
<pre>{{log}}</pre>` })
export class TextAreaComponent { textValue = 'initial value'; log = ''; logText(value: string):
void { this.log += `Text changed to '${value}'\n`; } }
```

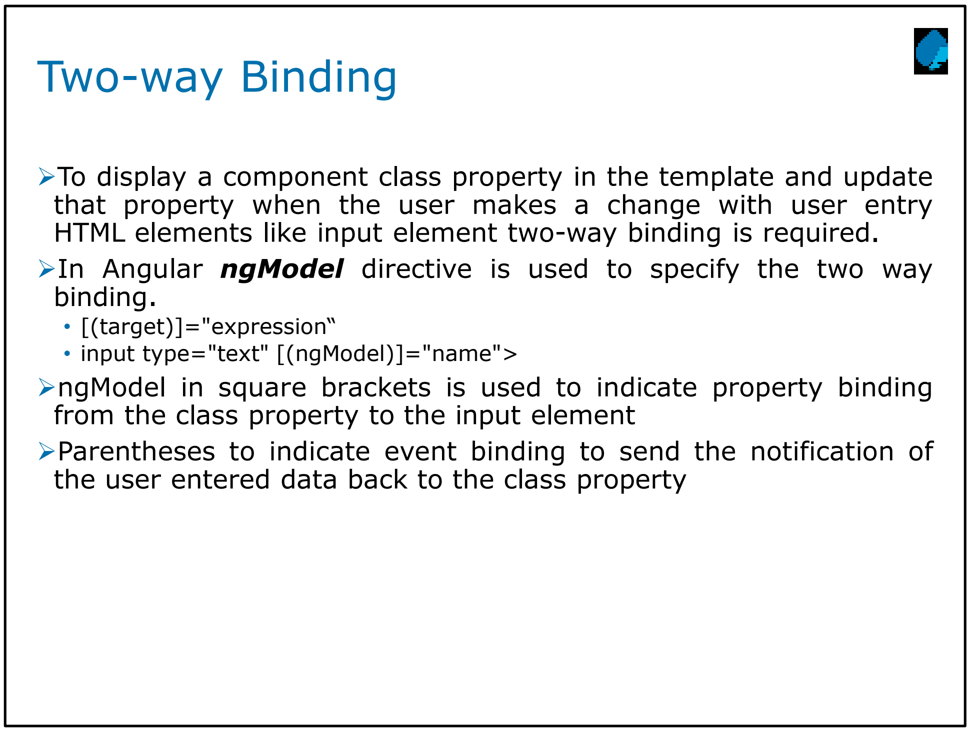
Radio

The radio field behaves in a similar way to the checkbox.

```
import { Component } from '@angular/core';
@Component({ selector: 'app-radio', template: ` <h1>Radio</h1> <p> <label
[class.selected]="r1.checked"> <input #r1 type="radio" name="r" value="one"
(change)="logRadio(r1)"> One </label> <label [class.selected]="r2.checked"> <input #r2
type="radio" name="r" value="two" (change)="logRadio(r2)"> Two </label> <label
[class.selected]="r3.checked"> <input #r3 type="radio" name="r" value="three"
(change)="logRadio(r3)"> Three </label> </p> <h2>Log <button
(click)="log=''">Clear</button></h2> <pre>{{log}}</pre>`, styles: ['.selected {color:
OrangeRed;}'] })
 export class RadioComponent { log = ''; logRadio(element: HTMLInputElement): void {
this.log += `Radio ${element.value} was selected\n`; } }
```

Demo

- Demo Event Binding



Two-way Binding

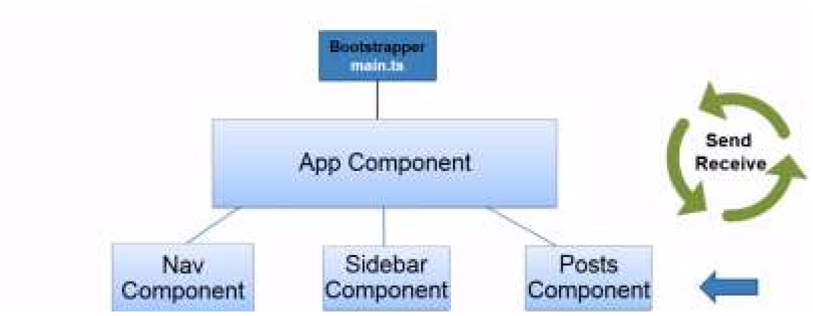
- To display a component class property in the template and update that property when the user makes a change with user entry HTML elements like input element two-way binding is required.
- In Angular ***ngModel*** directive is used to specify the two way binding.
  - [(target)]="expression‟
  - input type="text" [(ngModel)]="name">
- ngModel in square brackets is used to indicate property binding from the class property to the input element
- Parentheses to indicate event binding to send the notification of the user entered data back to the class property

Two-way data binding combines the square brackets of property binding with the parentheses of event binding in a single notation using the ngModel directive. Tip, to remember that the parentheses go inside the brackets, visualize a banana in a box. [()]
We bind to the ngModel property of the ngModel directive to update the contents of the input field. Conversely, the ngModelChange event takes care of setting of the value of the component property when the user changes the value in the text box.

Demo

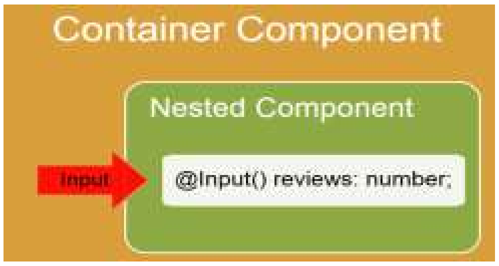
- Demo Two Way Binding

Nested Compoments



Using @Input and @Output

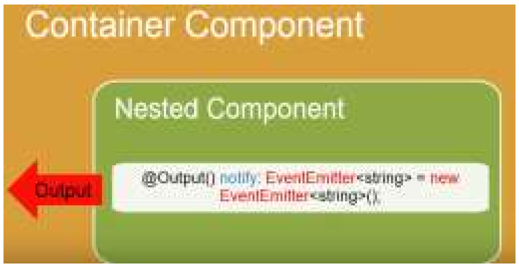
- @Input –Allows data to flow from parents component to child component
- @Input allows you to pass data into your controller and templates through html and defining custom properties.



Use property binding and the @Input decorator to pass data into a component, and @Output and EventEmitter to pass data out and bind to an event.

Using @Input and @Output (Contd…)

- @Output that pass data from child component to ParentComponent
- Components push out events using a combination of an @Output and an EventEmitter. This allows a clean separation between reusable Components and application logic.

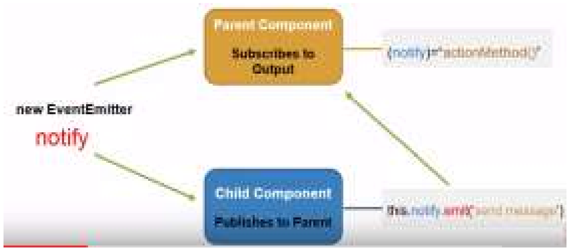


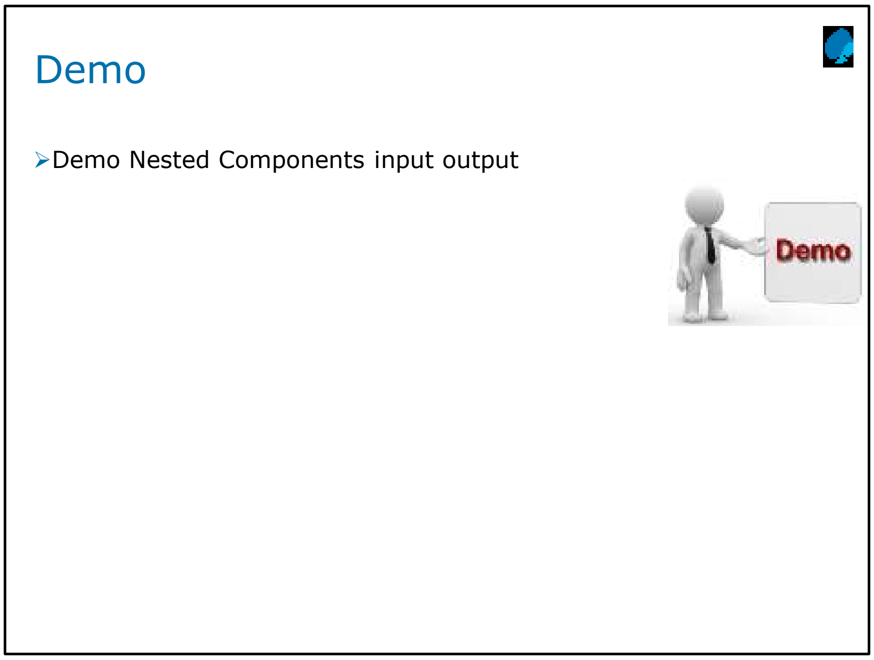
```
@Output() notify: EventEmitter<string> = new
EventEmitter<string>();
```

Event Binding with @Output
We can also pass values *out* of a child component back to the parent using an event binding. Let's see how.
In the child component we include an @Output decorator on a variable of type EventEmitter. This object will be used to fire our custom events.
@Output() colorValue: EventEmitter<string> = new EventEmitter();The variable name decorated by @Output is the name of the event, which in this case is colorValue.
The <string>type in angle brackets is the data type of the payload to be sent with the event.
We trigger the event by calling the emit method on the EventEmitter object. This method takes one argument which is the payload of the event.

Using @Input and @Output (Contd…)

- EventEmitter-Listen for something to happen & emit a event when triggered
- emit() method is used to trigger the event by emitting data from inner component to outer component which can be accessed via $event
- Nested component receives information from its container using input properties(@Input) and outputs information back to its container by raising events(@Output).

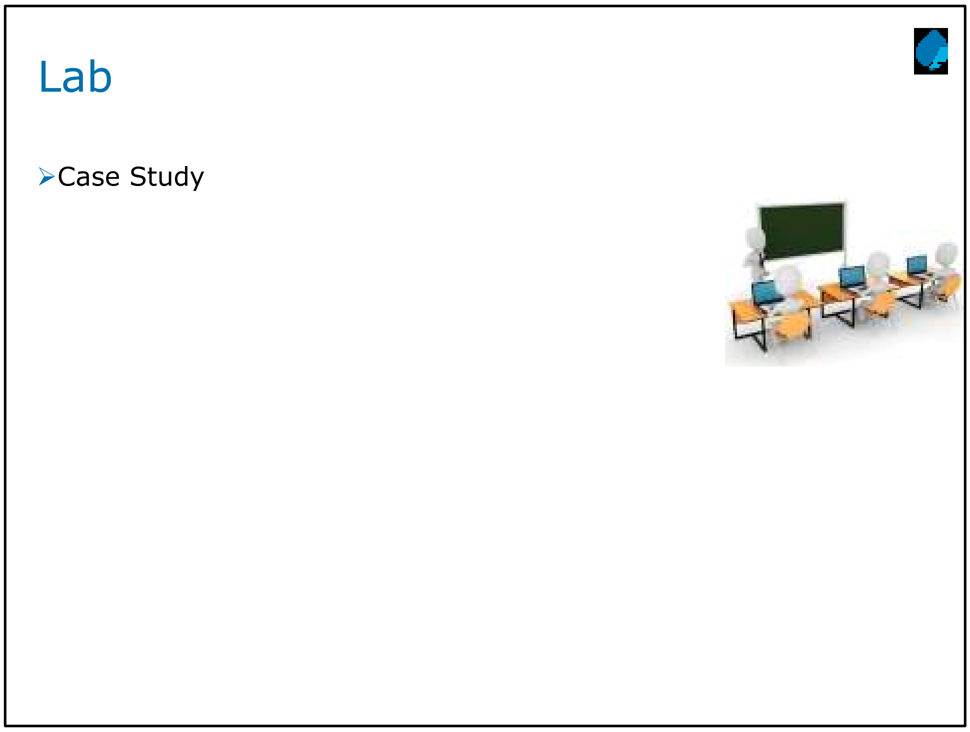




Demo

- ➢Demo Nested Components input output

Add instructor notes here.



Lab

- Case Study

Add the notes here.