CSM
breaking thought barriers

Angular Training

Session -19

Outlines

CSM
breaking thought barriers

- Decorator
- Angular Component
- Template VS TemplateUrl
- Nested Components
- Styling Components
- Databinding
- Angular Directives
- Angular Container and Nested Components
- Angular Pipes

Decorators

- Decorators are a way to decorate members of a class, or a class itself, with extra functionality.
- When you apply a decorator to a class or a class member, you are actually calling a function that is going to receive details of what is being decorated, and the decorator implementation will then be able to transform the code dynamically, adding extra functionality, and reducing boilerplate code.
- They are a way to have metaprogramming in TypeScript, which is a programming technique that enables the programmer to create code that uses other code from the application itself as data.

Enabling Decorators Support in TypeScript

Currently, decorators are still an experimental feature in TypeScript, and as such, it must be enabled first.

TypeScript Compiler CLI

```
tsc --experimentalDecorators
```

tsconfig.json

```
{
  "compilerOptions": {
    "experimentalDecorators": true
  }
}
```

Using Decorator Syntax

- A decorator is a function that is called with a specific set of parameters. These parameters are automatically populated by the JavaScript runtime and contain information about the class, method, or property to which the decorator has been applied.
- The number of parameters, and their types, determine where a decorator can be applied.

- To illustrate this syntax, let's define a class decorator as follows:

```
function simpleDecorator(constructor: Function) {
 console.log('simpleDecorator called');
}
```

This function, due to the parameters that it defines, can be used as a class decorator function and can be applied to a class definition as follows:

```
@simpleDecorator
class ClassWithSimpleDecorator {

}
let instance_1 = new ClassWithSimpleDecorator();
let instance_2 = new ClassWithSimpleDecorator();
console.log(`instance_1 : ${JSON.stringify(instance_1)}`);
console.log(`instance_2 : ${JSON.stringify(instance_2)}`);
```

OUTPUT :

```
simpleDecorator called
instance_1 : {}
instance_2 : {}
```

- ✓ We apply a decorator using the “at” symbol (@), followed by the name of the decorator function.
- ✓ **Note:** Decorators are only invoked once when a class is defined.

Multiple decorators

Multiple decorators can be applied one after another on the same target.

```
function secondDecorator(constructor: Function) {
 console.log(`secondDecorator called`);
}
```

```
@simpleDecorator
@secondDecorator
class ClassWithMultipleDecorators {
}
```

Note: Decorators are called in the reverse order of their appearance within our code.

Types of Decorators

- Decorators are functions that are invoked by the JavaScript runtime when a class is defined.
- Depending on what type of decorator is used, these decorator functions will be invoked with different arguments.

✓ **Class decorators:**
These are decorators that can be applied to a class definition.

✓ **Property decorators:**
These are decorators that can be applied to a property within a class.

✓ **Method decorators:**
These are decorators that can be applied to a method on a class.

✓ **Parameter decorators:**
These are decorators that can be applied to a parameter of a method within a class.

Example :

```
// Define a function called classDecorator which takes a constructor function as input
function classDecorator(
  constructor: Function
) {}
```

```
// Define a function called propertyDecorator which takes an object and a string property key as input
function propertyDecorator(
  target: any,
  propertyKey: string
) {}
```

Example :

*// Define a function called **methodDecorator** which takes an object, a string method name, and an optional property descriptor object as input*

```
function methodDecorator(
  target: any,
  methodName: string,
  descriptor?: PropertyDescriptor
) {}
```

*// Define a function called **parameterDecorator** which takes an object, a string method name, and a number representing a parameter index as input*

```
function parameterDecorator(
  target: any,
  methodName: string,
  parameterIndex: number
) {}
```

```
// Define a class called ClassWithAllTypesOfDecorators and apply the classDecorator to it
@classDecorator
class ClassWithAllTypesOfDecorators {
  // Apply the propertyDecorator to the id property of the class
  @propertyDecorator
  id: number = 1;

  // Apply the methodDecorator to the print method of the class
  @methodDecorator
  print() { }

  // Apply the parameterDecorator to the id parameter of the setId method of the class
  setId(@parameterDecorator id: number) { }
}
```

Class Decorators :

```
// Define a function called classConstructorDec which takes a constructor function as input
and logs it to the console
function classConstructorDec(constructor: Function) {
  console.log(`constructor : ${constructor}`);
}

// Apply the classConstructorDec decorator to the ClassWithConstructor class
@classConstructorDec
class ClassWithConstructor {
  constructor(id: number) { }
}
```

OUTPUT :

```
constructor : function ClassWithConstructor(id) {
  }
```

Property decorators:

```
// Define a function called propertyDec which takes an object and a string property name as input and logs them to the console
function propertyDec(target: any, propertyName: string) {
  console.log(`target : ${target}`);
  console.log(`target.constructor : ${target.constructor}`);
  console.log(`propertyName : ${propertyName}`);
}
// Define a ClassWithPropertyDec class and apply the propertyDec decorator to its nameProperty property
class ClassWithPropertyDec {
  @propertyDec
  nameProperty: string | undefined;
}
```

OUTPUT :

target : [object Object]

target.constructor : function ClassWithPropertyDec() {

}

propertyName : nameProperty

Method decorators:

```
// Define a methodDec function which logs the target, method name, descriptor, and target method
function methodDec(
  target: any,
  methodName: string,
  descriptor?: PropertyDescriptor
) {
  console.log(`target: ${target}`);
  console.log(`methodName : ${methodName}`);
  console.log(`descriptor : ${JSON.stringify(descriptor)}`);
  console.log(`target[methodName] : ${target[methodName]}`);
}

// Define a ClassWithMethodDec class and apply the methodDec decorator to its print method
class ClassWithMethodDec {
  @methodDec
  print(output: string) {
    console.log(`ClassWithMethodDec.print(${output}) called.`);
  }
}
```

Method decorators:

OUTPUT :

```
target: [object Object]
methodName : print
descriptor : {"writable":true,"enumerable":true,"configurable":true}
target[methodName] : function (output) {
        console.log("ClassWithMethodDec.print(".concat(output, ") called."));
    }
```

Parameter decorators:

```
function parameterDec(target: any,
 methodName: string,
 parameterIndex: number) {
 console.log(`target: ${target}`);
 console.log(`methodName : ${methodName}`);
 console.log(`parameterIndex : ${parameterIndex}`);
}

class ClassWithParamDec {
 print(@parameterDec value: string) {
 }
}

OUTPUT :
target: [object Object]
methodName : print
parameterIndex : 0
```

Angular Decorators

- Decorators are the features of Typescript and are implemented as functions.

- The name of the decorator starts with @ symbol following by brackets and arguments.

-The **@NgModule** decorator provides the necessary metadata to make the AppModule class as a module.

Note: If you want to create a module in angular, then you must decorate your class with @NgModule decorator. Once a class is decorated with @NgModule decorator, then only the class works as a module.

Commonly used Decorators:

@NgModule to define a module.
@Component to define components.
@Injectable to define services.
@Input and @Output to define properties

Note: All the above built-in decorators are imported from @angular/core library and so before using the above decorator, you first need to import the decorators from @angular/core library.

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

Types of Decorators in Angular:

1.Class Decorators: @Component and @NgModule

2.Property Decorators: @Input and @Output (These two decorators are used inside a class)

3.Method Decorators: @HostListener (This decorator is used for methods inside a class like a click, mouse hover, etc.)

4.Parameter Decorators: @Inject (This decorator is used inside class constructor).

Note: In Angular, each decorator has a unique role.

Angular Components

- According to Team Angular, A component controls a patch of screen real estate that we could call a view and declares reusable UI building blocks for an application.
- The core concept or the basic building block of Angular Application is nothing but the components. That means an angular application can be viewed as a collection of components and one component is responsible for handling one view or part of the view.
- An Angular Component encapsulates the data, the HTML Mark-up, and the logic required for a view.
- You can create as many components as required for your application.
- Every Angular application has at least one component that is used to display the data on the view.

- Technically, a component is nothing but a simple typescript class and composed of three things as follows:

- Class (Typescript class)
- Template (HTML Template or Template URL)
- Decorator (@Component Decorator)

Template:

- ✓ The template is used to define an interface with which the user can interact.
- ✓ As part of that template, you can define HTML Mark-up; you can also define the directives, and bindings, etc.
- ✓ The template renders the view of the application with which the end-user can interact i.e. user interface.

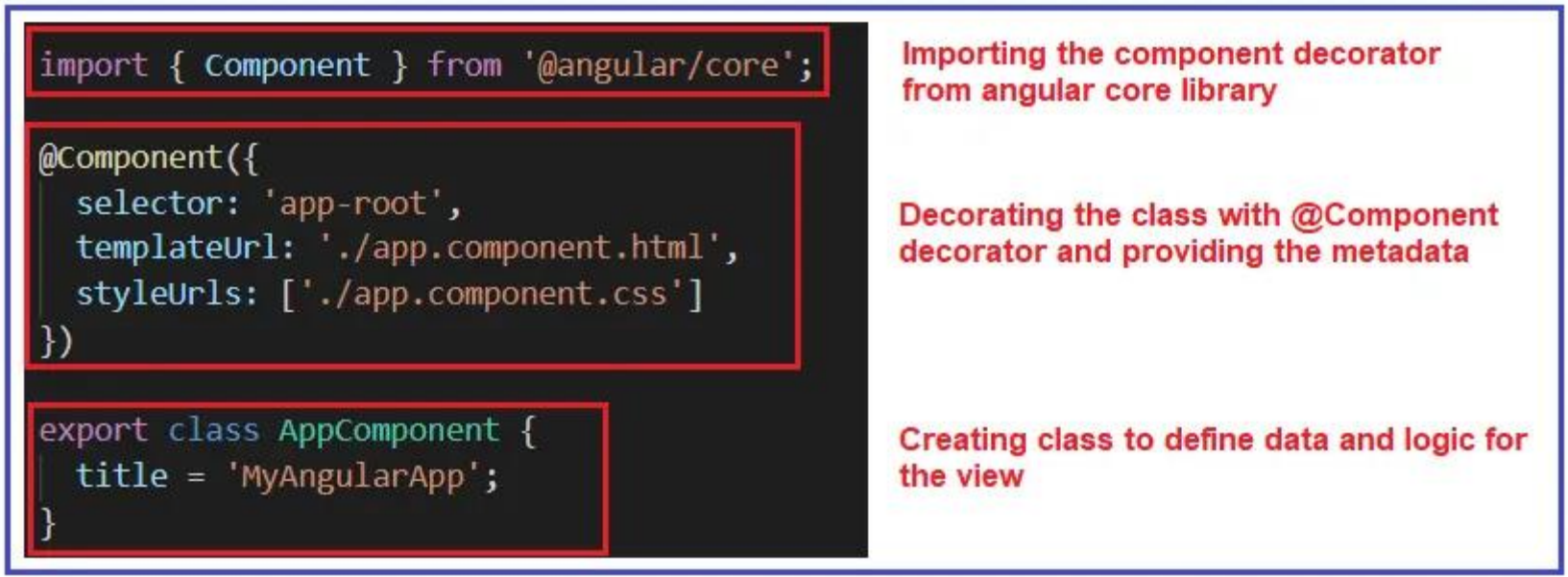
Class:

- ✓ The Class is the most important part of a component in which we can write the code which is required for a template to render in the browser.
- ✓ You can compare this class with any object-oriented programming language classes such as C++, C# or Java.
- ✓ The angular component class can also contain methods, variables, and properties like other programming languages.
- ✓ The angular class properties and variables contain the data which will be used by a template to render on the view.
- ✓ Similarly, the method in an angular class is used to implement the business logic like the method does in other programming languages.

Decorator:

- ✓ In order to make an angular class as a component, we need to decorate the class with the **@Component** decorator.
- ✓ Decorators are basically used to add metadata.

Note: Whenever we create any component, we need to define that component in @NgModule.


import { Component } from '@angular/core';
@Component({
selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']
})
export class AppComponent {
title = 'MyAngularApp';
}
Importing the component decorator from angular core library
Decorating the class with @Component decorator and providing the metadata
Creating class to define data and logic for the view

How to create a Component in Angular?

```
ng g c componentname
```

Template VS TemplateUrl in Angular

Different ways to create Templates in Angular

- Inline template (**template)**
- External Template (**templateUrl)**

Angular Nested Components

- ✓ The Angular framework allows us to use a component within another component and when we do so then it is called Angular Nested Components.

- ✓ The outside component is called the parent component and the inner component is called the child component.

Styling Angular Components

Option1: Component Inline Style

Option2: Component External Style

Option3: Template Inline Style using style tag

Option4: Template Inline Style using link tag

Option5: Global Style

Option6: ngClass and ngStyle

Component Inline Style

```
@Component({
  selector: 'app-test1',
  templateUrl: './test1.component.html',
  styles: [
    `p { color:blue}`,
     `h1 {color:blue}`
  ],
})
```

Component External Style

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html ',
  styleUrls: ['./app.component.css‘,
                  ,'.another.stylesheet.css'
                ]
})
```

Both Inline & External Style

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles:[`p {color:yellow}`],
  styleUrls: ['./app.component.css'],
})
```

Template Inline Style using style tag

```
<style>
  h1 {
    color: blue;
  }
  </style>

<h1>
  app works!
</h1>
```

Template Inline Style using link tag

```
<link rel="stylesheet" href="assets/css/morestyles.css">
<h1>
  app works!
</h1>
```

Global Styles

- Include global styles in the styles array of the angular.json file.
- This is useful for styles that need to be applied globally across the entire application.

```
"styles": [
  "src/styles.css",
  // Add other global stylesheets
]
```

NgClass Directive

- Include global styles in the styles array of the angular.json file.
- This is useful for styles that need to be applied globally across the entire application.

NgClass with a String

```
<element [ngClass]="'cssClass1 cssClass2'">...</element>
```

NgClass with Array

```
<element [ngClass]="['cssClass1', 'cssClass2']">...</element>
```

NgClass with Object

```
<element [ngClass]="{'cssClass1': true, 'cssClass2': true}">...</element>
```

ngStyle Directive

- The Angular ngStyle directive allows us to set the many inline style of a HTML element using an expression.
- The expression can be evaluated at run time allowing us to dynamically change the style of our HTML element.

ngStyle Syntax

```
<element [ngStyle]="{‘ styleNames': styleExp}">...</element>
```

```
<some-element [ngStyle]="{'font-size': '20px'}"></some-element>
```

```
color: string= 'red';
```

```
<div [ngStyle]="{'color': color, ‘font-size':20px}">Change my color</div>
```

Style Priority

The styles are applied in the following order

- **Component inline styles** i.e. Styles defined at @Component.styles
- **Component External styles** i.e. @Component.styleUrls
- **Template Inline Styles** using the style tag
- **Template External Styles** using the link tag

Add Bootstrap Library

Bootstrap is a popular front-end framework for building responsive web applications, and we can integrate it into an Angular application to quickly create stylish and responsive user interfaces.

Steps to install and use Bootstrap in an Angular project:

1. Install Bootstrap:

```
npm install bootstrap
```

2. Import Bootstrap CSS:

```
/* Add this line at the top of styles.css */
@import "~bootstrap/ dist / css /bootstrap.css";
```

3. Add Bootstrap JavaScript (Optional):

If you plan to use Bootstrap's JavaScript components, such as modals or carousels, you can import Bootstrap's JavaScript files into your project.

```
Angular.json

"scripts": [
  "node_modules /bootstrap/dist/js/bootstrap.js"
]
```

4. Use Bootstrap Components:

```
<!-- Example of using Bootstrap components in an Angular template -->
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="#">My Angular Bootstrap App</a>
  <!-- Add other Bootstrap components here -->
</nav>
```

Data Binding in Angular Application

Data binding is one of the most important features provided by Angular Framework which allows communicating between the component and its view.

Why do we need Data Binding?

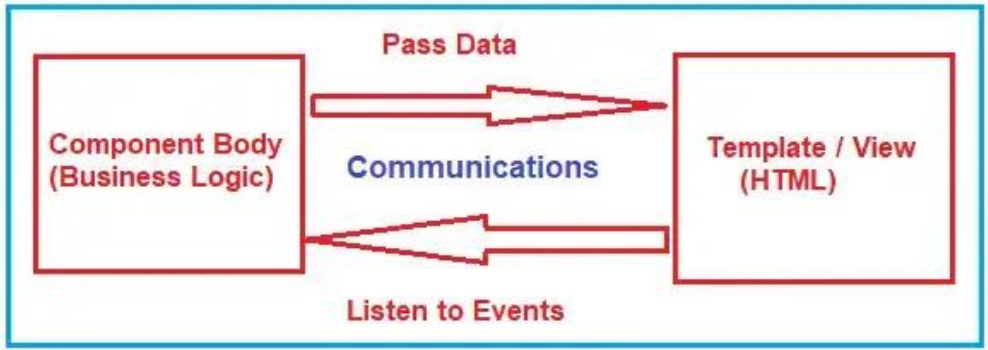
Whenever you want to develop any data-driven web application, then as a developer you need to keep the focus on two important things i.e. Data and the UI (User Interface) and it is more important for you to find an efficient way to bind them (Data and UI) together.

The angular framework provides one concept called Data Binding which is used for synchronizing the data and the user interface (called a view).

What is Data Binding in Angular Application?

In Angular, Data Binding means to bind the data (Component's filed) with the View (HTML Content). That is whenever you want to display dynamic data on a view (HTML) from the component then you need to use the concept Data binding.

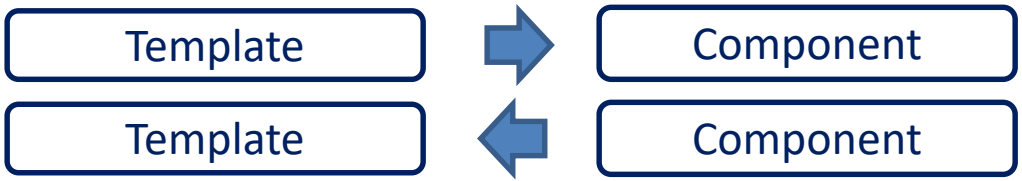
Data Binding is a process that creates a connection to communicate and synchronize between the user interface and the data. In order words, we can say that Data Binding means to interact with the data and view. So, the interaction between the templates (View) and the business logic is called data binding.



Types of Data Binding in Angular:

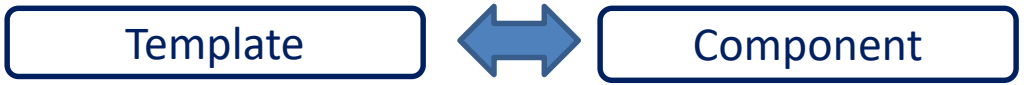
One-way Data Binding

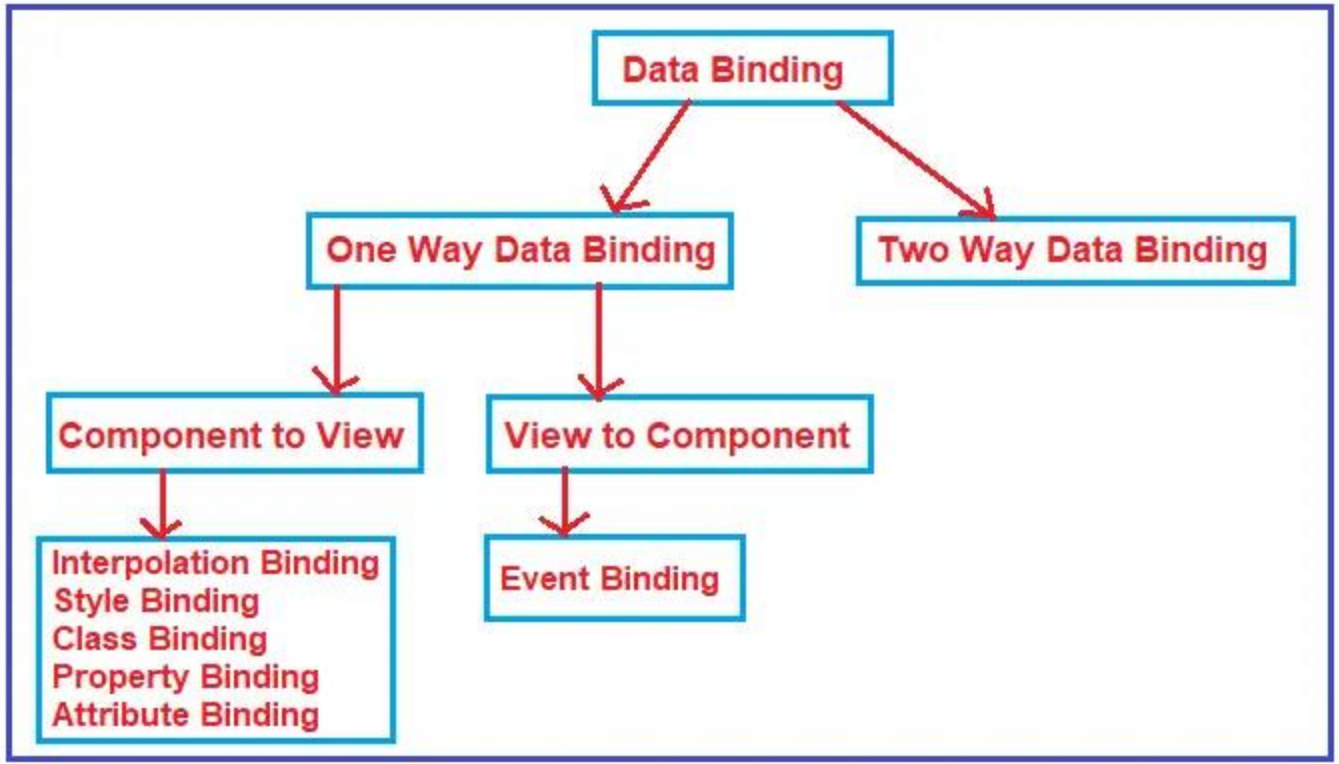
Where a change in the state affects the view (i.e. From Component to View Template) or change in the view affects the state (From View Template to Component).



Two-way Data Binding

Where a change from the view can also change the model and similarly change in the model can also change in the view (From Component to View Template and also From View template to Component).




Data Binding
One Way Data Binding
Two Way Data Binding
Component to View
View to Component
Interpolation Binding
Style Binding
Class Binding
Property Binding
Attribute Binding
Event Binding

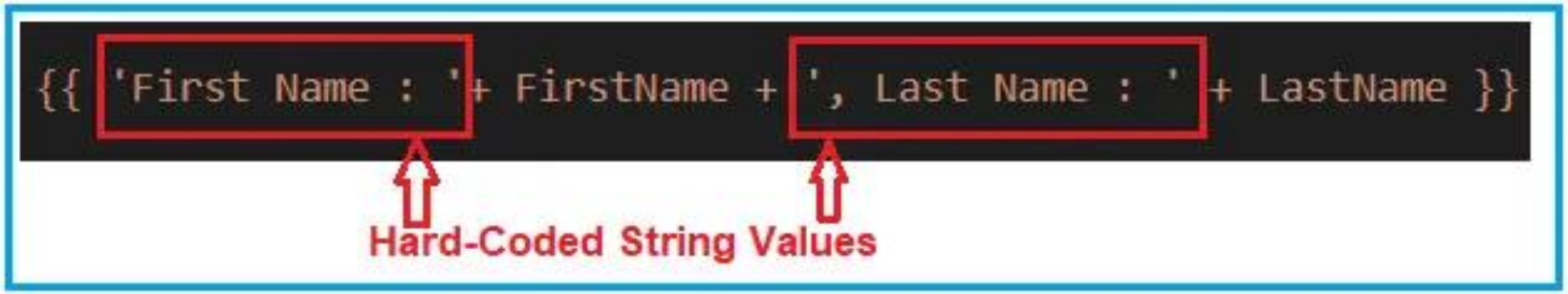
Examples of Angular Data Bindings:

- ❑ Interpolation
- ❑ Property Binding
- ❑ Attribute Binding
- ❑ Class Binding
- ❑ Style Binding
- ❑ Event Binding
- ❑ Two-way binding

Angular Interpolation

- If you want to display the read-only data on a view template (i.e. From Component to the View Template), then you can use the one-way data binding technique i.e. the Angular interpolation.
- Interpolation allows us to include expressions as part of any string literal, which we use in our HTML.
- You can use interpolation wherever you use a string literal in the view
- The Angular uses the {{ }} (double curly braces) in the template to denote the interpolation.
- Syntax : {{ templateExpression }}
- The content inside the double braces is called **Template Expression**
- The Angular first evaluates the Template Expression and converts it into a string. Then it replaces Template expression with the result in the original string in the HTML.
- Whenever the template expression changes, the Angular updates the original string again

Angular Interpolation with hardcoded string



Angular Interpolation with Expression:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
    template: `<div>
                <h1> Bonus = {{ Salary * .10 }} </h1>
              </div>`
})

export class AppComponent {
  Salary : number = 100000;
}
```

Interpolation in Angular with Ternary Operator:

```
import { Component } from '@angular/core';
@Component({
selector: 'app-root',
template: `<div>
<h1> Last Name : {{ LastName ? LastName : 'Not Available' }} </h1>
</div>`
})
export class AppComponent {
LastName : string = null;
}
```

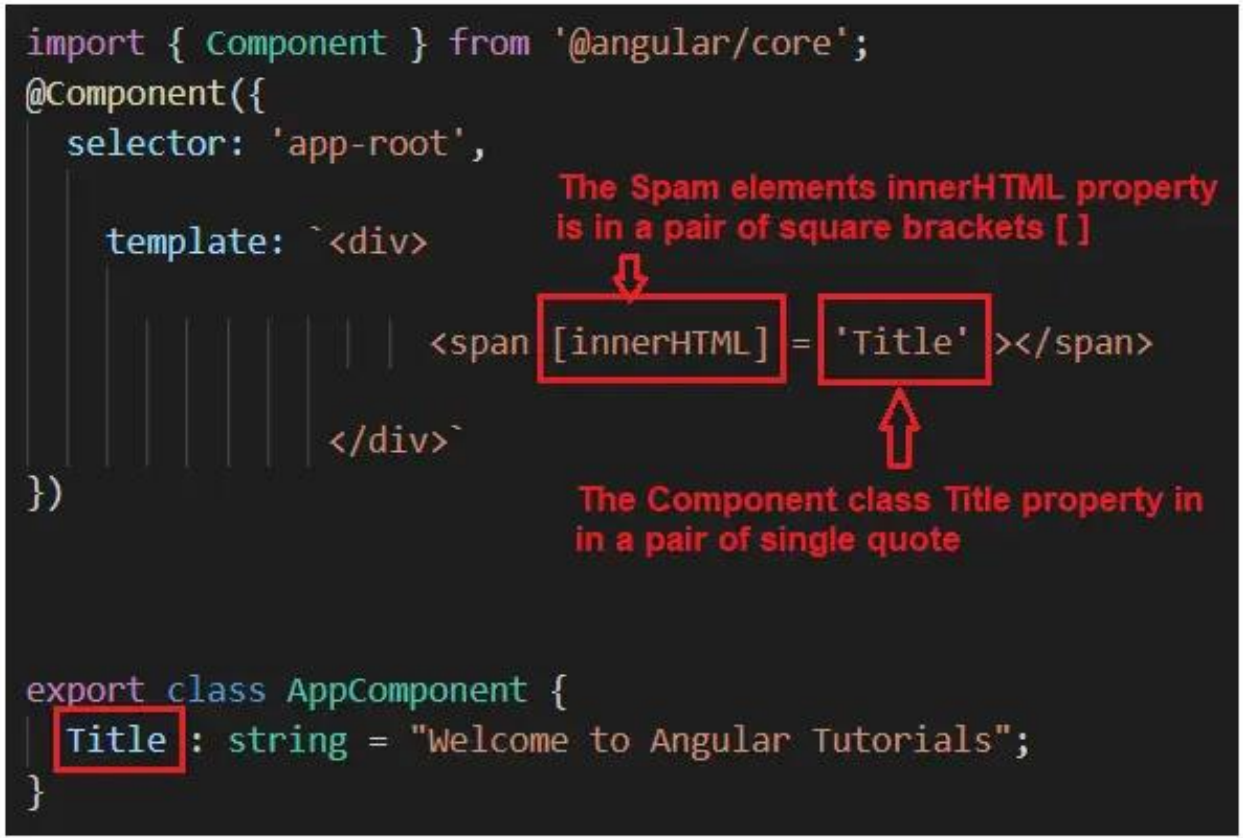
Method Interpolation in Angular Application:

```
{{ GetFullName() }}
```

Displaying Images using Angular Interpolation:

- The Property binding allows us to bind HTML element property to a property in the component.
- Whenever the value of the component changes, the Angular updates the element property in the View.
- We can set the properties such as class, href, src, textContent, etc using property binding.
- We can also use it to set the properties of custom components or directives (properties decorated with @Input).
- The Syntax :

  [binding-target]=”binding-source”
- The binding-target (or target property) is enclosed in a square bracket []. It should match the name of the property of the enclosing element.
- The Binding source must be a template expression. It can be property in the component, method in component, a template reference variable or an expression containing all of them.
- Example : **span[innerHTML] = ‘ FirstName’.**


import { Component } from '@angular/core';
@Component({
selector: 'app-root',
template: `
<span [innerHTML] = 'Title' >
`
})
The Spam elements innerHTML property is in a pair of square brackets [ ]
The Component class Title property in in a pair of single quote
export class AppComponent {
Title : string = "Welcome to Angular Tutorials";
}

Angular Interpolation and Property Binding

Interpolation in Angular is just an alternative approach for property binding. It is a special type of syntax that converts into a property binding.

Scenarios where we need to use interpolation instead of property binding :

1. **If you want to concatenate strings then you need to use angular interpolation instead of property binding**

Working with non-string (Boolean) data:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
    template: `<div>
                <button [disabled] = ‘ IsDisabledClick' > Click Here </button>
            </div>`
})
export class AppComponent {
  IsDisabledClick : boolean = true;
}
```

<button disabled = {{IsDisabledClick} **> Click Here </button>**

With the above changes in place, irrespective of the IsDisabledClick property value of the component class, the button is always disabled. Here we set the IsDisabled property value as false but when you run the application, it will not allow the button to be clickable.

Providing Security to Malicious Content:

From the security point of view, both Angular data binding and Angular Interpolation protect us from malicious HTML content before rendering it on the web browser.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
    template: `<div>
                  {{MaliciousData}}
              </div>`
})
export class AppComponent {
  MaliciousData : string = "Hello <script>alert('your application is hacked')</script>";
}
```

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
    template: `<div [innerHTML] = 'MaliciousData'>
              </div>`
})
export class AppComponent {
  MaliciousData : string = "Hello <script>alert('your application is hacked')</script>";
}
```

Both interpolation & property binding does not set the attributes of the HTML elements.

HTML Attribute VS DOM Property

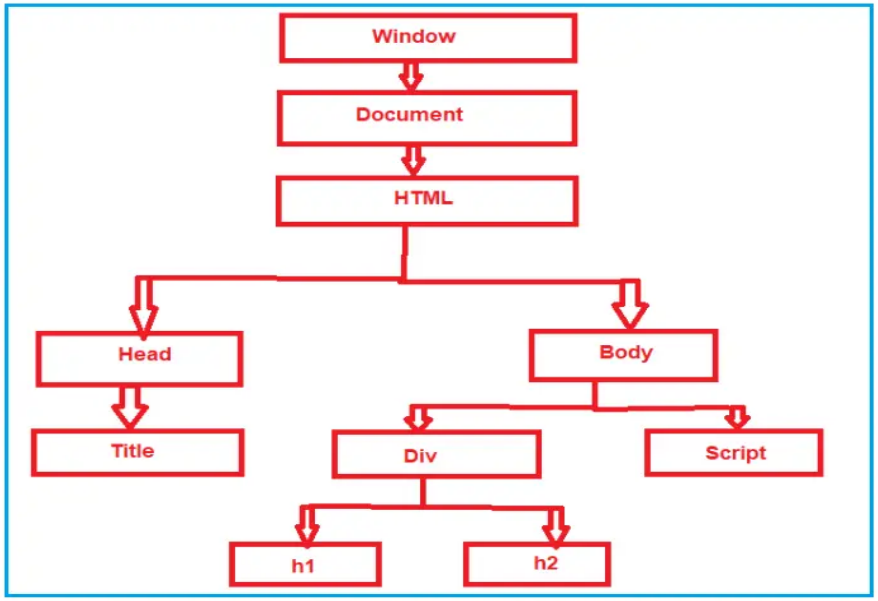
What is DOM?

The DOM stands for Document Object Model. When a browser loads a web page, then the browser creates the Document Object Model (DOM) for that page.

```
<html>
<head>
    <title>This is Title</title>
</head>
<body>
    <script src="Scripts/jquery-1.10.2.js"></script>
    <div>
        <h1>This is Browser DOM</h1>
        <h2>This is Inside H2</h2>
    </div>
</body>
</html>
```

The DOM is an application programming interface (API) for the HTML, and we can use the programming languages like JavaScript or JavaScript frameworks like Angular to access and manipulate the HTML using their corresponding DOM objects.

HTML Attribute VS DOM Property



DOM contains the HTML elements as objects, their properties, methods, and events and it is a standard for accessing, modifying, adding or deleting HTML elements.

Interpolation example: <button disabled='{{IsDisabled}}'>Click Me</button>

Property binding example: <button [disabled]='IsDisabled'>Click Me</button>

The Angular data-binding is all about binding to the DOM object properties and not the HTML element attributes.

What is the difference between the HTML element attribute and DOM property?

- The Attributes are defined by HTML whereas the properties are defined by the DOM.
- The attribute's main role is to initializes the DOM properties. So, once the DOM initialization complete, the attributes job is done.
- Property values can change, whereas the attribute values can never be changed.
- Angular binding works with the properties and events, and not with the attributes.

Example :

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
    template: `<div>
                <input id=‘ inputId ' type='text' value=‘ Alok '>
              </div>`
})
export class AppComponent {
}
```

Angular Attribute Binding

- In Angular Interpolation and Property Binding, we have seen that they both (Interpolation and Property Binding) are dealing with the DOM Properties but not with the HTML attributes.
- But there are some HTML elements (such as colspan, area, etc) that do not have the DOM Properties.
- With Attribute Binding in Angular, you can set the value of an HTML Element Attribute directly. So, the Attribute Binding is used to bind the attribute of an element with the properties of a component dynamically.

```
<thead>
    <tr>
        <th [attr.colspan]="ColumnSpan">
            {{pageHeader}}
        </th>
    </tr>
</thead>
```

```
<thead>
    <tr>
        <th attr.colspan={{ColumnSpan}}>
            {{pageHeader}}
        </th>
    </tr>
</thead>
```

Note: **The Angular team recommends using the property binding or Interpolation whenever possible and use the attribute binding only when there is no corresponding element property to bind.**

Angular Class Binding

The Angular Class Binding is basically used to add or remove classes to and from the HTML elements.

It is also possible in Angular to add CSS Classes conditionally to an element, which will create the dynamically styled elements and this is possible because of Angular Class Binding.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<div>
              <button [class] = ‘ClassesToApply' >Click Me</button>
            </div>`
})
export class AppComponent {
  ClassesToApply : string = ‘ italicClass boldClass';
}
```

If we want then we can also combine both class binding with the normal class

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<div>
              <button class='colorClass ' [class] = 'ClassesToApply' >Click Me</button>
            </div>`
})
export class AppComponent {
  ClassesToApply : string = 'italicClass boldClass';
}
```

Adding or removing a single class

If we want to add or remove a single class, then we need to use the prefix ‘class’ within a pair of square brackets and followed by a DOT (.) and the name of the class that you want to add or remove.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<div>
              <button class=‘ colorClass ' [ class.boldClass]='ApplyBoldClass'>Click Me</button>
            </div>`
})
export class AppComponent {
  ApplyBoldClass: boolean = true;
}
```

Angular Class Binding using “!” symbol:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<div>
              <button class='colorClass' [class.boldClass]='!ApplyBoldClass'>Click Me</button>
            </div>`
})
export class AppComponent {
  ApplyBoldClass: boolean = false;
}
```

Add or Remove multiple classes in Angular:

In order to add or remove multiple style classes in angular, the angular framework provides one directive called **ngClass directive** which we can use to remove or add multiple classes

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<div>
              <button class='colorClass' [ngClass]='AddCSSClasses() '>Click Me</button>
            </div>`
})
export class AppComponent {
    ApplyBoldClass: boolean = true;
    ApplyItalicsClass: boolean = true;
    AddCSSClasses() {
        let Cssclasses = {
            boldClass: this.ApplyBoldClass,
            italicsClass: this.ApplyItalicsClass
        };
        return Cssclasses;
    }
}
```

Angular Style Binding

- The Angular Style Binding is basically used to set the style in HTML elements.
- We can use both inline as well as Style Binding to set the style in the element in Angular Applications.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<div>
              <button style='color:red ' [ style.font-weight]="IsBold ? 'bold' : 'normal'">Click Me</button>
            </div>`
})
export class AppComponent {
  IsBold: boolean = true;
}
```

Note : The style property name can be written in either dash-case or camelCase.

Some styles like font-size have a unit extension. To set the font-size in pixels, we need to use the following syntax.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<div>
              <button style='color:red' [style.font- size.px ]="FontSize">Click Me
              </button>
            </div>`

})
export class AppComponent {
  FontSize: number = 40;
}
```

Multiple Inline Styles in Angular Application:

If we want to set multiple inline styles in the angular application, then you need to use NgStyle directive

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<div>
              <button style='color:red' [ngStyle]="AddCSSStyles()">Click Me </button>
            </div>`
})
export class AppComponent {
    IsBold: boolean = true;
    FontSize: number = 40;
    IsItalic: boolean = true;
    AddCSSStyles() {
        let CssStyles = {
            'font-weight': this.IsBold ? 'bold' : 'normal',
            'font-style': this.IsItalic ? 'italic' : 'normal',
            'font-size.px': this.FontSize
        };
        return CssStyles;
    }
}
```

Angular Event Binding

When a user interacts with an application in the form of a keyboard movement, button click, mouse over, selecting from a drop-down list, typing in a textbox, etc. it generates an event. These events need to be handled to perform some kind of action.

How Does Event Binding work in Angular?

```
<button (click)="onClick()">Click Me </button>
```

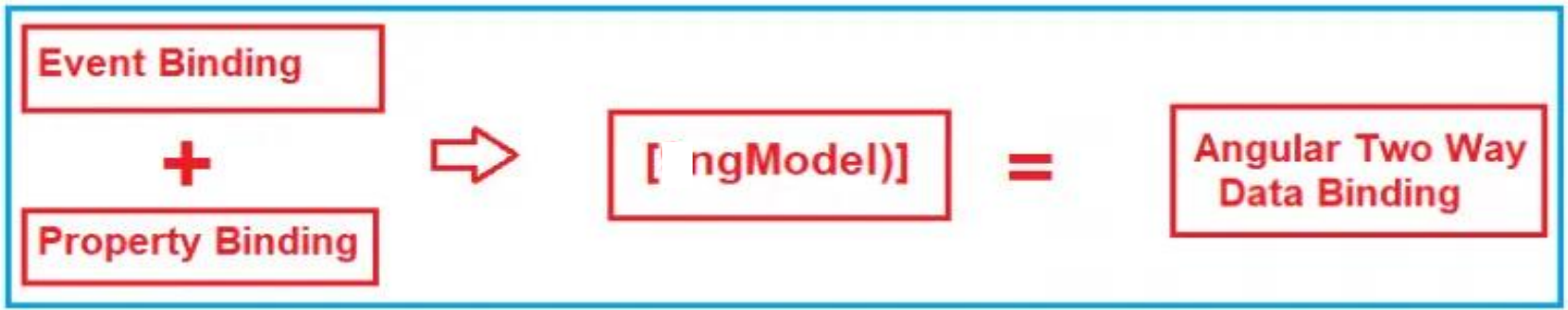
With event binding, you can also use the on- prefix alternative as shown below .This is known as the canonical form.

```
<button on-click="onClick()">Click Me </button>
```

Angular Two Way Binding

The most popular and widely used data binding mechanism in Angular Application is two-way data binding.

The two-way data binding is basically used in the input type field or any form element where the user type or provide any value or change any control value on the one side and on the other side, the same automatically updated into the component variables and vice-versa is also true.



The two-way data binding in Angular is actually a combination of Property Binding and Event Binding.

The Syntax is given below:

```
<input [value] = ‘data’ (input) = ‘data = $event.target.value’>
```

Two-Way Binding using ngModel Directive:

- The ngModel directive combines the square brackets of property binding with the parentheses of event binding in a single notation.

- The syntax to use ngModel for two-way data binding is given below.

```
<input [(ngModel)] = 'data'>
```

```
Name : <input [value]='Name' (input) = 'Name = $event.target.value'>

Change to ⇩

Name : <input [(ngModel)]='Name' >
```

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<div>
            Name : <input  [(ngModel)]=‘Name’>
            <br>
            You entered : {{Name}}
           </div>`
})
export class AppComponent {
  Name: string = ‘ Alok ';
}
```

```
ERROR in src/app/app.component.ts:7:29 - error NG8002: Can't bind to 'ngModel' since it
isn't a known property of 'input'.

7           Name : <input [(ngModel)]='Name'>
```

Steps to use ngModel Directive:

1. Open app.module.ts file

2. Include the following import statement in it
   import { FormsModule } from '@angular/forms';

3. Also, include FormsModule in the 'imports' array of @NgModule
   imports: [BrowserModule, FormsModule]

Case Study : Product Management System

Requirements:

1.Display Product List:

Display a list of products with their names, prices, and quantities.

2.Add New Product:

Allow users to add new products to the system with a name, price, and initial quantity.

3.Edit Product Details:

Enable users to edit the details (name, price, quantity) of existing products.

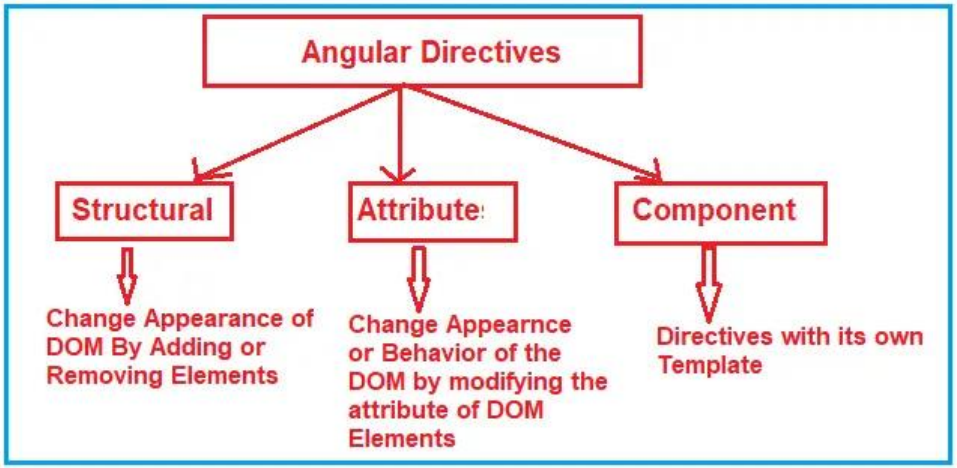
4.Delete Product:

Provide the ability to delete products from the system.

Angular Directives

- The Angular directive helps us to manipulate the DOM. You can change the appearance, behavior, or layout of a DOM element using the Directives.

- There are three kinds of directives in Angular:



Structural Directives:

Structural directives can change the DOM layout by adding and removing DOM elements. All structural Directives are preceded by Asterix symbol

NgIf (*ngIf)

NgSwitch (*ngSwitch)

NgFor (*ngFor)

Attribute Directives:

- Attribute Directives are basically used to modify the behavior or appearance of the DOM element or the Component.

1. **NgStyle:** This NgStyle Attribute Directive is basically used to modify the element appearance or behavior.

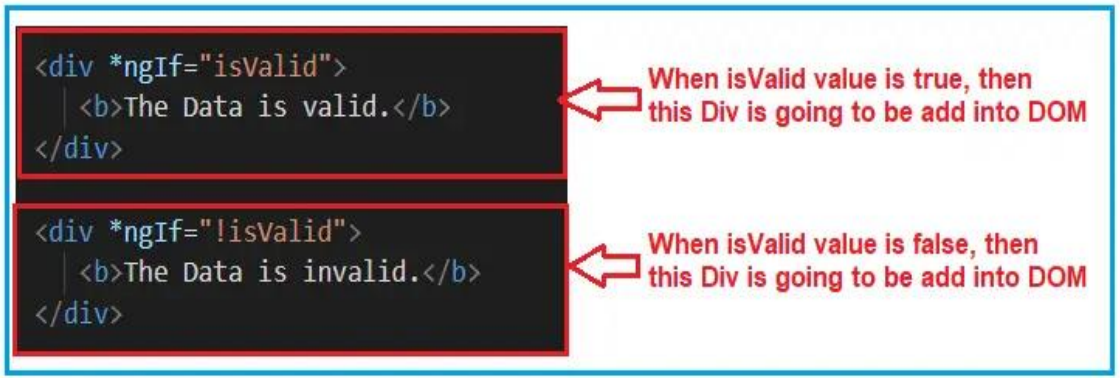
2. NgClass : This NgClass Attribute Directive is basically used to change the class attribute of the element in the DOM or in the Component to which it has been attached.

Component Directives

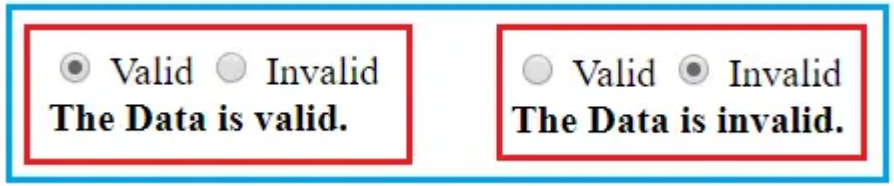
- Components are special directives in Angular. They are the directive with a template (view)
- The Component is also a type of directive in angular with its own template, styles, and logic needed for the view.
- The Component Directive is the most widely used directive in the angular application and you cannot create an angular application without a component.
- A component directive requires a view along with its attached behavior and this type of directive adds DOM Elements.
- The Component Directive is a class with @Component decorator function.
- The naming convention for components is name.component.ts.

Angular ngIf Directive

- The ngIf is a structural directive and it is used to add or removes the HTML element and its descendant elements from the DOM layout at runtime conditionally.
- The ngIf directive works on the basis of a boolean true and false results of a given expression. If the condition is true, the elements will be added into the DOM layout otherwise they simply removed from the DOM layout.
- The syntax : ***ngIf = "expression"**



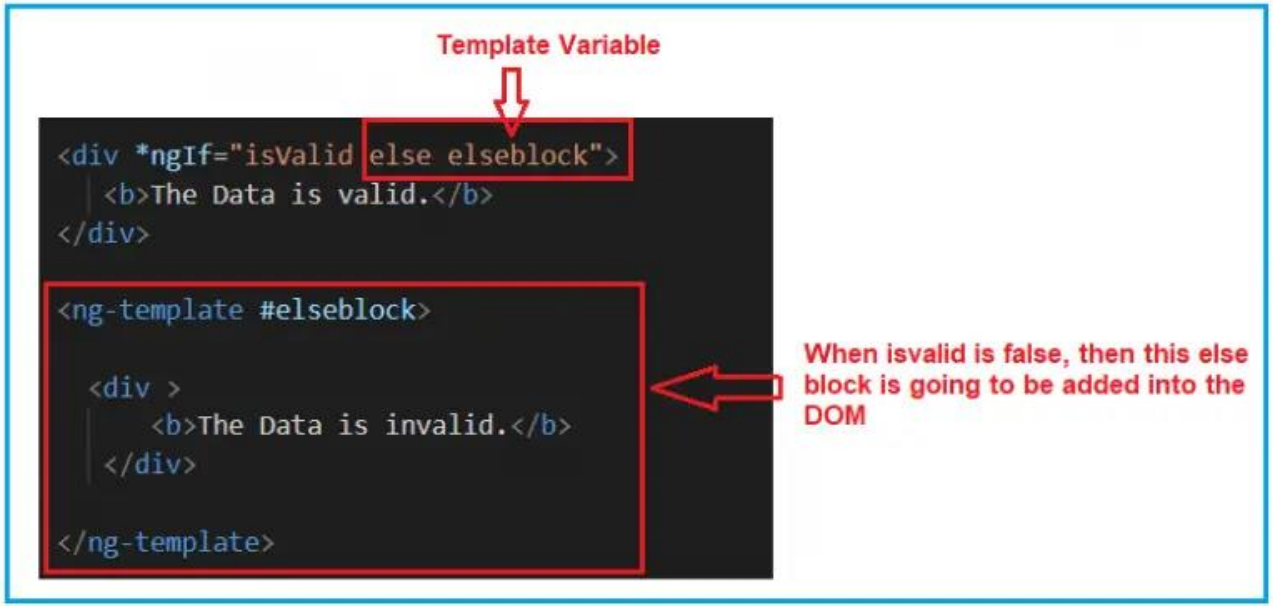
Example :


Valid Invalid
The Data is valid.
Valid Invalid
The Data is invalid.

Angular NgIf directive with else block:

<div *ngIf = “condition; else elseBlock”>…

<ng-template #elseblock>….</ng-tempalte>



NgIf with Then and else:

```
<div *ngIf="condition; then thenBlock else elseBlock"></div>
<ng-template #thenBlock>...</ng-template>
<ng-template #elseBlock>...</ng-template>
```

```
<div *ngIf="isValid then thenblock else elseblock"> </div>
<ng-template #thenblock>
<div>
<b>The is Then Block</b>
</div>
</ng-template>

<ng-template #elseblock>
<div >
<b>The is Else Block</b>
</div>
</ng-template>
```

Angular ngSwitch Directive

- ✓ The Angular ngSwitch directive is actually a combination of two directives i.e. an attribute directive and a structural directive.
- ✓ The ngSwitch directive lets you add/remove HTML elements depending on a match expression. ngSwitch directive used along with ngSwitchCase and ngSwitchDefault

ngSwitch

ngSwitchCase

ngSwithDefault

```
<div [ngSwitch]="Switch_Expression">
   <div *ngSwitchCase="MatchExpression1”> First Template</div>
   <div *ngSwitchCase="MatchExpression2">Second template</div>
   <div *ngSwitchCase="MatchExpression3">Third Template</div>
   <div *ngSwitchCase="MatchExpression4">Third Template</div>
   <div *ngSwitchDefault?>Default Template</div>
</div>
```


Select Country
Select
No Country code is selected
You Have Selected
You have not selected any country


Select Country
In
You Have Selected
India

Angular ngFor Directive

- ✓ The ngFor is an Angular structural directive, which repeats a portion of the HTML template once per each item from an iterable list (Collection).

- ✓ The syntax : ***ngFor=”let <value> of <collection>”**

- ✓
```
<tr *ngFor="let customer of customers;">
        <td>{{customer.customerNo}}</td>
        <td>{{customer.name}}</td>
        <td>{{customer.address}}</td>
        <td>{{customer.city}}</td>
        <td>{{customer.state}}</td>
    </tr>
```

Example : Display Employees

ngFor – Local Variables:

1. **Index:** This variable is used to provide the index position of the current element while iteration.
2. **First:** It returns boolean true if the current element is the first element in the iteration else it will return false.
3. **Last:** It returns boolean true if the current element is the last element in the iteration else it will return false.
4. **Even:** It returns boolean true if the current element is even element based on the index position in the iteration else it will return false.
5. **Odd:** It returns boolean true if the current element is an odd element based on the index position in the iteration else it will return false.

Angular ngFor trackBy

- The use of trackBy is to improve the performance of the angular application.
- It is usually not needed by default but needed only when application running into performance issues.
- The Angular ngFor directive may perform poorly with the large collections. A small change to the collection such as adding a new item or removing an existing item from the collection may trigger a cascade of DOM manipulations.

Angular ngFor trackBy

- The use of trackBy is to improve the performance of the angular application.
- It is usually not needed by default but needed only when application running into performance issues.
- The Angular ngFor directive may perform poorly with the large collections. A small change to the collection such as adding a new item or removing an existing item from the collection may trigger a cascade of DOM manipulations.

Attribute Directives

ngStyle Directive

- The Angular ngStyle directive allows us to set the many inline style of a HTML element using an expression.
- The expression can be evaluated at run time allowing us to dynamically change the style of our HTML element.
- Syntax : **<element [ngStyle]="{‘styleNames': styleExp}">...</element>**

- Example :

```
<p [ngStyle]="{'font-size': '20px'}">Set Font size to 20px</p>
<p [ngStyle]="{'font-size.em': '3'}">...</p>
```

- **Change Style Dynamically**

```
color: string= 'red';
<input [(ngModel)]="color" />
<div [ngStyle]="{'color': color}">Change my color</div>
<div [ngStyle]="{'background-color ': status === 'error' ? 'red' : 'blue' }"></<div>
```

- **ngStyle multiple attributes**

```
<p [ngStyle]="{'color': 'purple',
            'font-size': '20px',
            'font-weight': 'bold'}">
    Multiple styles
</p>
```

- **Specifying CSS Units in ngStyle**

CSS has several units for expressing a length, size etc. The units can be em, ex, %, px, cm, mm, in, pt, PC etc.

```
<input [(ngModel)]="size" />
<div [ngStyle]="{'font-size.px': size}">Change my size</div>
```

- **Using object from Controller**

```
class StyleClass {
   'color': string= 'blue';
   'font-sizeclass StyleClass {
   'color': string= 'blue';
   'font-size.px': number= 20;
   'font-weight': string= 'bold';
}.px': number= 20;
   'font-weight': string= 'bold';
}
```

```
styleClass: StyleClass = new StyleClass();
```

```
<div [ngStyle]="styleClass">size & Color</div>
```

ngClass Directive

- Its allows us to add or remove CSS classes to an HTML element.
- ngClass makes adding a conditional class to an element easier, hence dynamically changing the styles at runtime.
- **NgClass with a String**

  <element [ngClass]="'cssClass1 cssClass2'">...</element>
- We can also use the ngClass without a square bracket. In that case, the expression is not evaluated but assigned directly to the class attribute.

  Sample Text
- **NgClass with Array**

  <element [ngClass]="['cssClass1', 'cssClass2']">...</element>
- **NgClass with Object**

  <element [ngClass]="{'cssClass1': true, 'cssClass2': true}">...</element>

ngClass Directive

- **Applying class from component**
- **Conditionally applying class**

```
numbers = [30, 40, 50, 60, 70, 80]
 getClass(num) {
            if (num <= 50) return 'primary';
            else return 'secondary';
          }

<ul>
   <li *ngFor="let num of numbers">
     <div [ngClass]="getClass(num)">{{ num }}</div>
    <div [ngClass]="{ primary: num <= 50, secondary: num > 50 }">
      {{ num }}
     </div>

   </li>
 </ul>
```

Custom Directives

1.Create a custom directive using the @Directive decorator.

2.We will create both custom attribute directive & custom Structural directive

3.How to setup selectors

4.Pass value to it using the @input.

5.How to respond to user inputs,

6.Manipulate the DOM element (Change the Appearance) etc.

.

Creating Custom Attribute Directive

1. Create a new file and name it as **tt-class.directive.ts.** import the necessary libraries that we need.

```
import { Directive, ElementRef, Input, OnInit } from '@angular/core‘
```

2. Decorate the class with @Directive

```
@Directive({
 selector: '[ttClass]',
})
export class ttClassDirective implements OnInit {
}
```

3. Our directive needs to take the class name as the input. The Input decorator marks the property ttClass as the input property. It can receive the class name from the parent component.

```
@Input() ttClass: string;
```

Creating Custom Attribute Directive

4. - We attach the attribute directive to an element, which we call the parent element.
   - To change the properties of the parent element, we need to get the reference.
   - Angular injects the parent element when we ask for the instance of the ElementRef in its constructor.

```
constructor(private el: ElementRef) {
}
```

- ElementRef is a wrapper for the Parent DOM element. We can access the DOM element via the property **nativeElement**.

- The classList method allows us to add the class to the element.

```
ngOnInit() {
    this.el.nativeElement.classList.add( this.ttClass);
}
```

Creating Custom Attribute Directive

```
import { Directive, ElementRef, Input, OnInit } from '@angular/core'

@Directive({
  selector: '[ttClass]',
})
export class ttClassDirective implements OnInit {

  @Input() ttClass: string;

  constructor(private el: ElementRef) {
  }

  ngOnInit() {
    this.el.nativeElement.classList.add( this.ttClass);
  }

}
```

Creating Custom Attribute Directive

app.component.css

```
.blue {
  background-color: lightblue;
}

<button [ttClass]="'blue'">Click Me</button>
```

Custom Directive in Angular

To create a custom directive, we'll typically work with attribute or structural directives.

Creating a custom attribute directive in Angular:

1. Create the Directive:

```
ng generate directive my-custom-directive
```

2. Implement the Directive:

3. Use the Directive:

```
<div appMyCustomDirective>
  This div has the custom directive applied.
</div>
```

4. include your custom directive in your Angular module's declarations array

```
import { Directive, ElementRef } from
'@angular/core';

@Directive({
  selector: '[appMyCustomDirective]'
})
export class MyCustomDirectiveDirective {

  constructor(private el: ElementRef) {
    el.nativeElement.style.backgroundColor =
'yellow';
  }

}
```

Creating a custom structural directive in Angular

1. ng generate directive my-structural-directive

2. Implement the Directive:

3. Use the Directive:

```
<div *appMyStructuralDirective="true">
  This content will be shown if the condition is true.
</div>

<div *appMyStructuralDirective="false">
  This content will be removed if the condition is false.
</div>
```

4. include your custom directive in your Angular module's declarations array.

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';
@Directive({
  selector: '[appMyStructuralDirective]'
})
export class MyStructuralDirectiveDirective {
  @Input() set appMyStructuralDirective(condition: boolean) {
    if (condition) {
      this.viewContainer.createEmbeddedView( this.templateRef);
    } else {
      this.viewContainer.clear();
    }
  }
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef
  ) { }
}
```

Multiple user events, like mouse enter and inputs, can also be handled in custom directives using **@HostListener** and **@Input** tags, respectively

```
import { Directive, ElementRef, HostListener, Input } from "@angular/core";
@Directive({
  selector: "[appAlterBackgroundHandler]"
})
export class AlterBackgroundHandlerDirective {
  constructor(private el: ElementRef) { }

  @Input() appAlterBackgroundHandler = "";

  @HostListener("mouseenter") onMouseEnter() {
    this.changeBgColor( this.appAlterBackgroundHandler );
  }
  @HostListener("mouseleave") onMouseLeave() {
    this.changeBgColor("blue");
  }
  private changeBgColor( color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}
```

```
<!-- handling events and inputs -->
<div>
  <h3 [appAlterBackgroundHandler]="'green'">Handling User Inputs and Events</h3>
</div>
```

Angular Container and Nested Components

Component Communication

1.Parent to Child Communication

2.Child to Parent Communication

3.Interaction when there is no parent-child relation

Parent to Child Communication

- If the Components have a parent-child relationship then, then the parent component can pass the data to the child using the @input Property.
- **Using @Input Decorator to Pass Data**

  Create a property (someProperty) in the Child Component and decorate it with @Input(). This will mark the property as input property

```
export class ChildComponent {
    @Input() someProperty: number;
}
```

- In the Parent Component Instantiate the Child Component. Pass the value to the someProperty using the Property Bind Syntax

```
<child-component [someProperty]=value></child-component>
```

How to Pass data to a child component

- In Angular, the Parent Component can communicate with the child component by setting its Property.

- To do that the Child component must expose its properties to the parent component. The Child Component does this by using the @Input decorator

- **In the Child Component :**
  - Import the @Input module from @angular/Core Library
  - Mark those property, which you need data from the parent as input property using @Input decorator

- **In the Parent Component :**
  Bind the Child component property in the Parent Component when instantiating the Child

@Input Decorator

- The @Input Decorator is used to configure the input properties of the component.
- This decorator as also supports change tracking.
- When you mark a property as input property, then the Angular injects values into the component property using Property Binding.
- The Property Binding uses the [] brackets.

```
import { Component, Input  } from '@angular/core';
@Component({
   selector: 'child-component',
   template: `<h2>Child Component</h2>
            current count is {{ count }}
   `
})
export class ChildComponent {
   @Input() count: number;
}
```

Bind to Child Property in Parent Component

```
import { Component} from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
       <h1>Welcome to {{title}}!</h1>
       <button (click)="increment()">Increment</button>
       <button (click)="decrement()">decrement</button>
       <child-component [count]=Counter></child-component>` ,
        styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Component Interaction';
  Counter = 5;

  increment() {
    this.Counter++;
  }
  decrement() {
    this.Counter--;
  }
}
```

Various ways to use @Input Decorator

- We used input @Input Decorator to mark the property in child component as input property.
- There are two ways you can do it Angular.
  1. Using the @Input decorator to decorate the class property

```
export class ChildComponent {
    @Input() count: number;
}
```

  2. Using the input array meta data of the component decorator

```
@Component({
   selector: 'child-component',
   inputs: ['count'],
   template: `<h2>Child Component</h2>
   current count is {{ count }}
`
})
export class ChildComponent {}
```

Aliasing input Property

```
export class ChildComponent {
    @Input(‘ MyCount ') count: number;
}
```

In the parent component, we can use the MyCount as shown below template:

```
<h1>Welcome to {{title}}!</h1>
<child-component [MyCount]=ClickCounter></child-component>
```

Detecting the Input changes

- Passing the data to child component is not sufficient, the child Component needs to know when the input changes so that it can act upon it.

- There are two ways of detecting when input changes in the child component in Angular

  1. Using OnChanges LifeCycle Hook
  2. Using Input Setter

1. Using OnChanges LifeCycle Hook

- **ngOnChanges** is a lifecycle hook, which angular fires when it detects changes to data-bound input property.
- This method receives a **SimpeChanges** object, which contains the current and previous property values. We can Intercept input property changes in the child component using this hook.

- **How to use ngOnChanges for Change Detection**

1. Import the **OnChanges** interface,**SimpleChanges,SimpleChange** from @angular/core library.

2. Implement the **ngOnChanges()** method. The method receives the **SimpleChanges** object containing the changes in each input property.

1. Using OnChanges LifeCycle Hook

```
import { Component, Input, OnChanges, SimpleChanges, SimpleChange  } from '@angular/core';
@Component({
   selector: 'child-component',
   template: `<h2>Child Component</h2>
            current count is {{ count }}`
})
export class ChildComponent implements OnChanges {
   @Input() count: number;
   ngOnChanges(changes: SimpleChanges) {
      for (let property in changes) {
         if (property === 'count') {
           console.log('Previous:', changes[property].previousValue);
           console.log('Current:', changes[property].currentValue);
           console.log('firstChange:', changes[property].firstChange);
         }
      }
   }
}
```

1. Using OnChanges LifeCycle Hook

- This method receives all the changes made to the input properties as SimpleChanges object.
- The SimpleChanges object whose keys are property names and values are instances of SimpleChange.
- SimpleChange class Represents a basic change from a previous to a new value. It has three class members.

Property Name	Description
previousValue:any	Previous value of the input property.
currentValue:any	New or current value of the input property.
FirstChange():boolean	Boolean value, which tells us whether it was the first time the change has taken place

2. Using Input Setter

- We can use the property getter and setter to detect the changes made to the input property.

- In the Child Component create a private property called _count

  private _count = 0;

- Create getter & setter on property count and attach @Input Decorator. We intercept the input changes from the setter function.

```
@Input()
set count(count: number) {
   this._count = count;
   console.log(count);
}
get count(): number { return this._count; }
```

Child to Parent Communication

The Child to Parent communication can happen in three ways.

1. Listens to Events from Child
2. Uses Local Variable to access the child in the Template
3. Uses a @ViewChild to get a reference to the child component

Listens to Child Event

- This is done by the child component by exposing an **EventEmitter** Property.
- We also decorate this Property with @Output decorator.
- When Child Component needs to communicate with the parent it **raises the emit event** of the **EventEmitter** Property.
- The Parent Component listens to that event and reacts to it.
- **EventEmitter Property**
  - To Raise an event, the component must declare an EventEmmitter Property.
  - The Event can be emitted by calling the .emit() method
  - Example :

    countChanged: EventEmitter<number> = new EventEmitter()

  - Then call emit method passing the whatever the data you want to send

    this.countChanged.emit(this.count);

@Output Decorator

Using the EventEmitter Property gives the components ability to raise an event. But to make that event accessible from parent component, you must decorate the property with @Output decorator

How to Pass data to parent component using @Output ?

In the child component

- Declare a property of type EventEmitter and instantiate it
- Mark it with a @Output Decorator
- Raise the event passing it with the desired data

In the Parent Component

- Bind to the Child Component using Event Binding and listen to the child events
- Define the event handler function

Example :

Place a counter to the child component. Then raise an event in the child component whenever the count is increased or decreased. Then bind to that event in the parent component and display the count in the parent component.

child.component.ts :

```
import { Component, Input, Output, EventEmitter  } from '@angular/core';
 @Component({
   selector: 'child-component',
   template: `<h2>Child Component</h2>
            <button (click)="increment()">Increment</button>
            <button (click)="decrement()">decrement</button>
            current count is {{ count }}
   `
})
export class ChildComponent {
   @Input() count: number;
    @Output() countChanged: EventEmitter<number> =   new EventEmitter();
    increment() {
      this.count++;
      this.countChanged.emit(this.count);
     }
   decrement() {
      this.count--;
      this.countChanged.emit(this.count);
   }
}
```

Parent Component

```
import { Component} from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
       <h1>Welcome to {{title}}!</h1>
       <p> current count is {{ClickCounter}} </p>
       <child-component [count]=Counter (countChanged)="countChangedHandler($event)"></child-component>`
,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Component Interaction';
  Counter = 5;

  countChangedHandler(count: number) {
    this.Counter = count;
    console.log(count);
  }
}
```

Parent uses local variable to access the Child in Template

Parent Template can access the child component properties and methods by creating the template reference variable

Child Component

```
import { Component} from '@angular/core';
 @Component({
   selector: 'child-component',
   template: `<h2>Child Component</h2>
            current count is {{ count }}
   `
})
export class ChildComponent {
   count = 0;
    increment() {
      this.count++;
     }
   decrement() {
      this.count--;
   }
}
```

Parent component

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

@Component({
  selector: 'app-root',
  template: `
       <h1>{{title}}!</h1>
       <p> current count is {{child.count}} </p>
       <button (click)="child.increment()">Increment</button>
       <button (click)="child.decrement()">decrement</button>
       <child-component #child></child-component>` ,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Parent interacts with child via local variable';
}
```

The “child” is called template reference variable, which now represents the child component

Parent uses a @ViewChild() to get reference to the Child Component

- Injecting an instance of the child component into the parent as a @ViewChild is the another technique used by the parent to access the property and method of the child component

- The @ViewChild decorator takes the name of the component/directive as its input. It is then used to decorate a property. The Angular then injects the reference of the component to the Property

Child Component

```
import { Component} from '@angular/core';
 @Component({
    selector: 'child-component',
    template: `<h2>Child Component</h2>
              current count is {{ count }}
    `
})
export class ChildComponent {
    count = 0;
     increment() {
        this.count++;
      }
    decrement() {
        this.count--;
    }
}
```

Parent Component

```
import { Component, ViewChild } from '@angular/core';
import { ChildComponent } from './child.component';
 @Component({
  selector: 'app-root',
  template: `
       <h1>{{title}}</h1>
       <p> current count is {{child.count}} </p>
       <button (click)="increment()">Increment</button>
       <button (click)="decrement()">decrement</button>
       <child-component></child-component>` ,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Parent calls an @ViewChild()';

  @ViewChild(ChildComponent) child: ChildComponent;

  increment() {
    this.child.increment();
  }

  decrement() {
    this.child.decrement();
  }
}
```

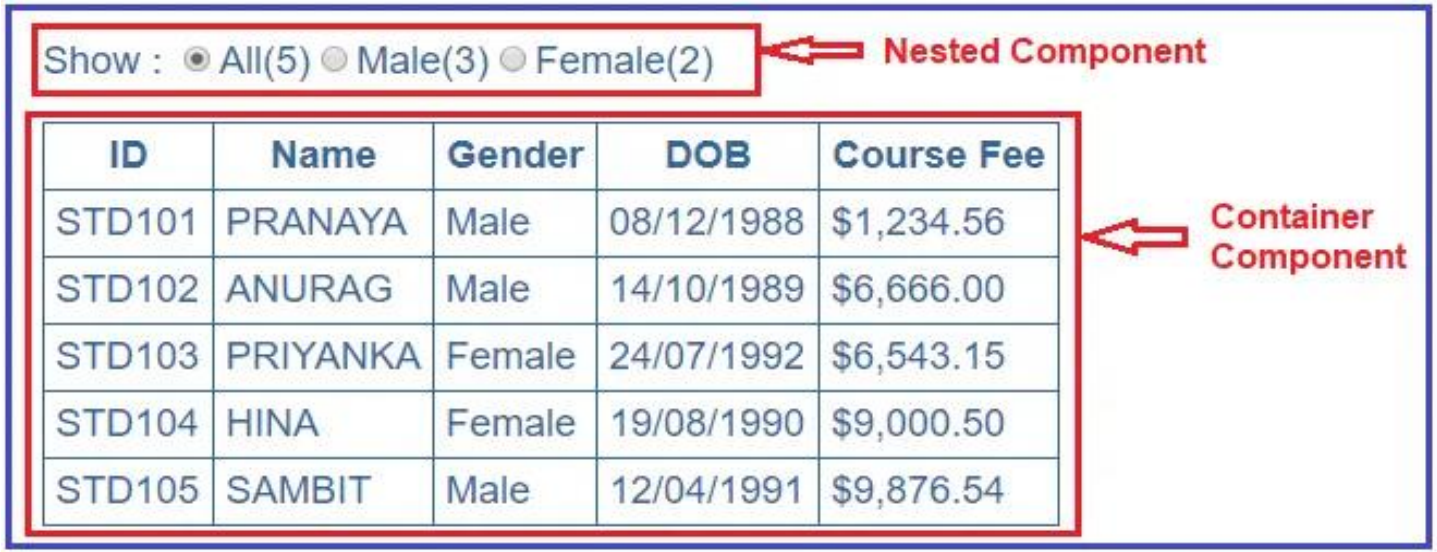
Exercise :

Show : ◉ All(5) ○ Male(3) ○ Female(2)

ID	Name	Gender	DOB	Course Fee
STD101	PRANAYA	Male	08/12/1988	$1,234.56
STD102	ANURAG	Male	14/10/1989	$6,666.00
STD103	PRIYANKA	Female	24/07/1992	$6,543.15
STD104	HINA	Female	19/08/1990	$9,000.50
STD105	SAMBIT	Male	12/04/1991	$9,876.54

StudentListComponet

StudentCountComponet



Angular Component Input Properties

The Angular Component Input Properties are used to pass the data from container component to the nested component

Angular Component Output Properties

The Angular Component Output Properties are used to to pass the data from the nested component to the container component.

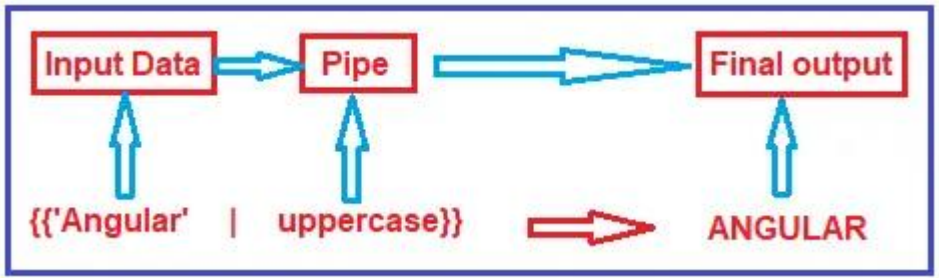
Angular Pipes

Angular Pipes

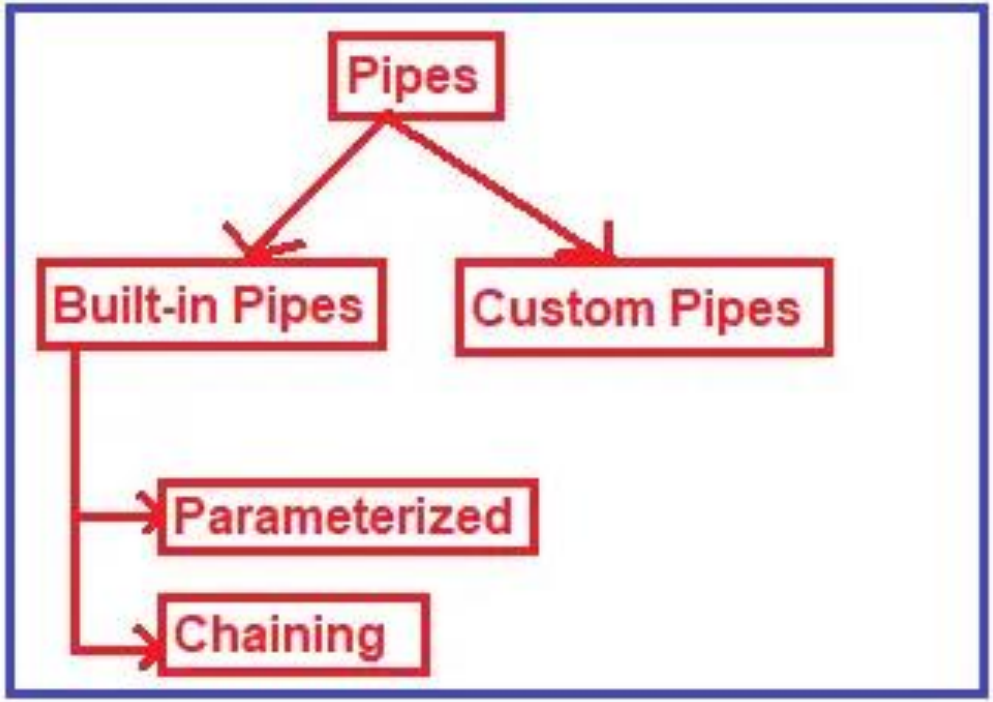
- The angular pipes transform the data into a specific format before displaying them to the end-users.

- Using the Pipe (|) operator, we can apply the pipes features to any of the property in angular application.

Syntax to use Pipes in Angular Application:



Types of Pipes in Angular:



Angular Parameterized Pipes

In Angular, we can pass any number of parameters to the pipe using a colon (:) and when we do so, it is called Angular Parameterized Pipes.

The syntax :

Syntax:

```
data | pipeName : parameter 1 : parameter 2 : parameter 3 ... parameter n
```

Examples:

Date:- `{{DOB | date : "short"}}`

Currency:- `{{courseFee | currency : 'USD' : true : '1.3-3'}}`

Date Pipe:

```
today: number = Date.now();
```

```
<p>Date Pipe : {{today | date}}</p>
<p>Full Date : {{today | date:'fullDate'}}</p>
<p>Mediate Date : {{today | date:'medium'}}</p>
<p>Short Date : {{today | date:'short'}}</p>
<p>Date (dd/MM/yyyy) : {{today |
date:'dd/MM/yyyy'}}</p>
<p>Time : {{today | date:'h:mm a z'}}</p>
<p>Medium Time : {{today | date:'mediumTime'}}</p>
```

Currency Pipe:

```
<p>Currency USD in Symbol : {{salary | currency:'USD':true}}</p>
<p>Currency INR in Symbol : {{salary | currency:'INR':true}}</p>

<p>Currency USD in Code : {{salary | currency:'USD':false:'4.2-2'}}</p>
<p>Currency INR in Code : {{salary | currency:'INR':false:'1.3-3'}}</p>
```

JSON Pipe:

```
{{ object | json }}
```

Slice Pipe:

```
{{ array | slice:1:5 }}
```

Angular Custom Pipe

ng g pipe MyTitle –flat

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: ‘myTitle '
})
export class MyTitlePipe implements PipeTransform {
  transform(name: string, gender: string): string {
    if (gender.toLowerCase() == "male")
       return "Mr. " + name;
    else
       return "Miss. " + name;
  }
}
```

```
<td>{{student.Name | myTitle:student.Gender}}</td>
```

Angular lifecycle hooks

- When the angular application starts, it creates and renders the root component. It then creates and renders its Children & their children.

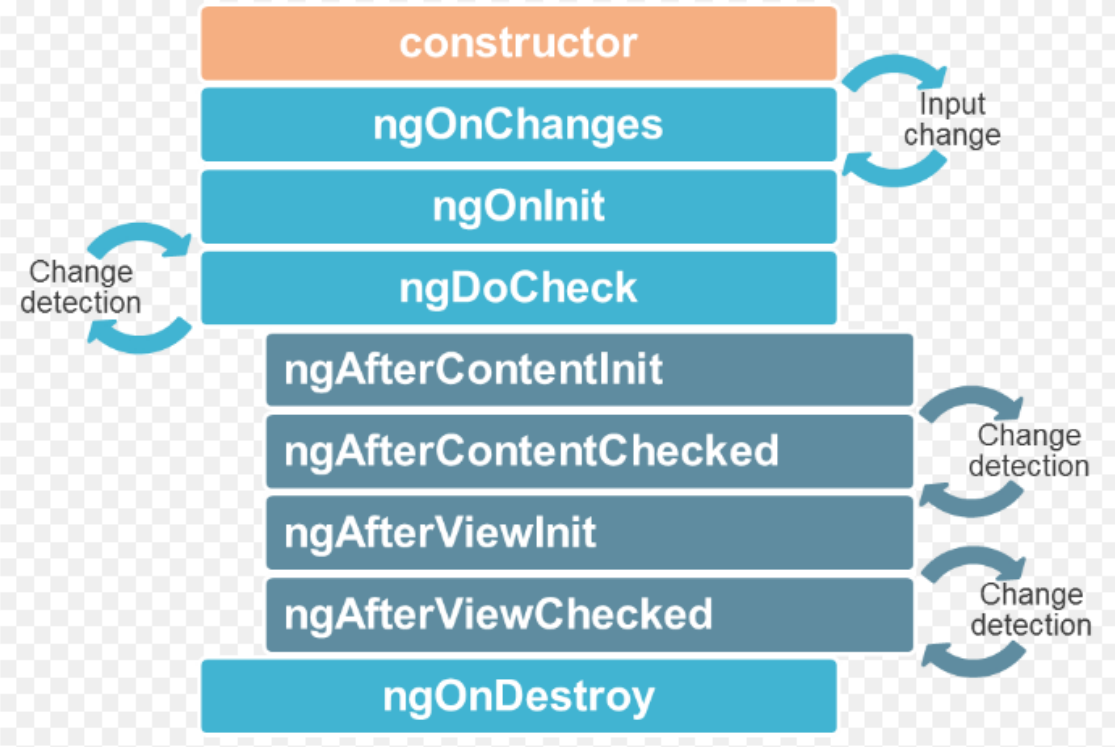
- Once Angular loads the components, it starts rendering the view. To do that, it needs to check the input properties, evaluate the data bindings & expressions, render the projected content, etc. Angular also removes the component from the DOM when it no longer needs it.

- Angular lets us know when these events happen using lifecycle hooks

- The Angular life cycle hooks are nothing but callback functions, which angular invokes when a specific event occurs during the component's life cycle.

Angular lifecycle hooks

1. ngOnChanges
2. ngOnInit
3. ngDoCheck
4. ngAfterContentInit
5. ngAfterContentChecked
6. ngAfterViewInit
7. ngAfterViewChecked
8. ngOnDestroy



Change detection Cycle

- Change detection is the mechanism by which angular keeps the template in sync with the component
- Example: Hello {{name}}
- Angular updates the DOM whenever the value of the name changes. And it does it instantly.

How does angular know when the value of the name changes?

- It does so by running a change detection cycle on every event that may result in a change.
- It runs on every input change, DOM event, and timer event like setTimeout(), setInterval(), HTTP requests, etc.

During the change detection cycle, angular checks every bound property in the template with that of the component class. If it detects any changes, it updates the DOM.

Angular raises the life cycle hooks during the critical stages of the change detection mechanism.

Constructor

- The life cycle of a component begins when Angular creates the component class. The first method that gets invoked is class Constructor.
- Constructor is neither a life cycle hook nor is it specific to Angular. It is a Javascript feature. It is a method that is invoked when a class is created.
- Angular makes use of a constructor to inject dependencies.
- At this point, none of the component’s input properties are available. Neither its child components are constructed.
- Hence there is little you can do with this method. And also, it is recommended not to use it.
- Once Angular instantiates the class, It kick-starts the first change detection cycle of the component.

ngOnChanges

- The Angular invokes the ngOnChanges life cycle hook whenever any data-bound input property of the component or directive changes.
- Initializing the Input properties is the first task angular carries during the change detection cycle. And if it detects any change in property, then it raises the **ngOnChanges** hook.
- It does so during every change detection cycle. This hook is not raised if change detection does not detect any changes.
- The change detector checks if the parent component changes such input properties of a component. If it is, then it raises the **ngOnChanges** hook.

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

@Component({
   selector: 'child-component',
   template: `<h2>Child Component</h2>
            current count is {{ count }}
   `
})
export class ChildComponent implements OnChanges {
   @Input() count: number;

   ngOnChanges(changes: SimpleChanges) {

      for (let property in changes) {
         if (property === 'count') {
           console.log('Previous:', changes[property].previousValue);
           console.log('Current:', changes[property].currentValue);
           console.log('firstChange:', changes[property].firstChange);
         }
      }
   }
}
```

ngOnInit

- The Angular raises the **ngOnInit** hook after it creates the component and updates its input properties. It raises it after the **ngOnChanges** hook.
- This hook is fired **only once** and immediately after its creation (during the first change detection).
- This is a perfect place where you want to add any initialization logic for your component. Here you have access to every input property of the component.
- You can use them in **HTTP get requests** to get the data from the back-end server or run some initialization logic etc.
- But note that none of the child components are available at this juncture.
- Hence any properties we decorate with @ViewChild, @ViewChildren, @ContentChild & @ContentChildren will not be available to use.

ngDoCheck

- The Angular invokes the ngDoCheck hook event during every change detection cycle.
- This hook is invoked even if there is no change in any of the properties.

- Angular invoke it after the ngOnChanges & ngOnInit hooks.

- Use this hook to Implement a custom change detection whenever Angular fails to detect the changes made to Input properties. This hook is convenient when you opt for the Onpush change detection strategy.

ngDoCheck

- The Angular invokes the ngDoCheck hook event during every change detection cycle.
- This hook is invoked even if there is no change in any of the properties.

- Angular invoke it after the ngOnChanges & ngOnInit hooks.

- Use this hook to Implement a custom change detection whenever Angular fails to detect the changes made to Input properties. This hook is convenient when you opt for the Onpush change detection strategy.

- The Angular ngOnChanges hook does not detect all the changes made to the input properties.

ngAfterContentInit

- **ngAfterContentInit** Life cycle hook is called after the Component's projected content has been fully initialized.

- Angular also updates the properties decorated with the ContentChild and ContentChildren before raising this hook. This hook is also raised, even if there is no content to project.

- The content here refers to the external content injected from the parent component via **Content Projection**.

ngAfterContentChecked

- **ngAfterContentChecked** Life cycle hook is called during every change detection cycle after Angular finishes checking of component's projected content.
- Angular also updates the properties decorated with the ContentChild and ContentChildren before raising this hook.
- Angular calls this hook even if there is no projected content in the component.
- This hook is very similar to the **ngAfterContentInit** hook. Both are called after the external content is initialized, checked & updated. The only difference is that **ngAfterContentChecked** is raised after every change detection cycle. While **ngAfterContentInit** during the first change detection cycle.

ngAfterViewInit

- **ngAfterViewInit** hook is called after the Component's View & all its child views are fully initialized.
- Angular also updates the properties decorated with the ViewChild & ViewChildren properties before raising this hook.
- The View here refers to the template of the current component and all its child components & directives.
- This hook is called during the first change detection cycle, where angular initializes the view for the first time.
- At this point, all the lifecycle hook methods & change detection of all child components & directives are processed & Component is entirely ready.

ngAfterViewChecked

- The Angular fires this hook after it checks & updates the component's views and child views. This event is fired after the **ngAfterViewInit** and after that, during every change detection cycle.
- This hook is very similar to the **ngAfterViewInit** hook. Both are called after all the child components & directives are initialized and updated. The only difference is that **ngAfterViewChecked** is raised during every change detection cycle. While **ngAfterViewInit** during the first change detection cycle.

ngOnDestroy

- This hook is called just before the Component/Directive instance is destroyed by Angular
- You can Perform any cleanup logic for the Component here. This is where you would like to Unsubscribe Observables and detach event handlers to avoid memory leaks.

How to Use Lifecycle Hooks

1. **Import Hook interfaces**
2. **Declare that Component/directive Implements lifecycle hook interface**
3. **Create the hook method**

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

@Component({
  selector: 'app-root',
  template: `
      <h2>Life Cycle Hook</h2>` ,
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {

  constructor() {
    console.log("AppComponent:Constructor");
  }

  ngOnInit() {
    console.log("AppComponent:OnInit");
  }
}
```

The Order of Execution of Life Cycle Hooks

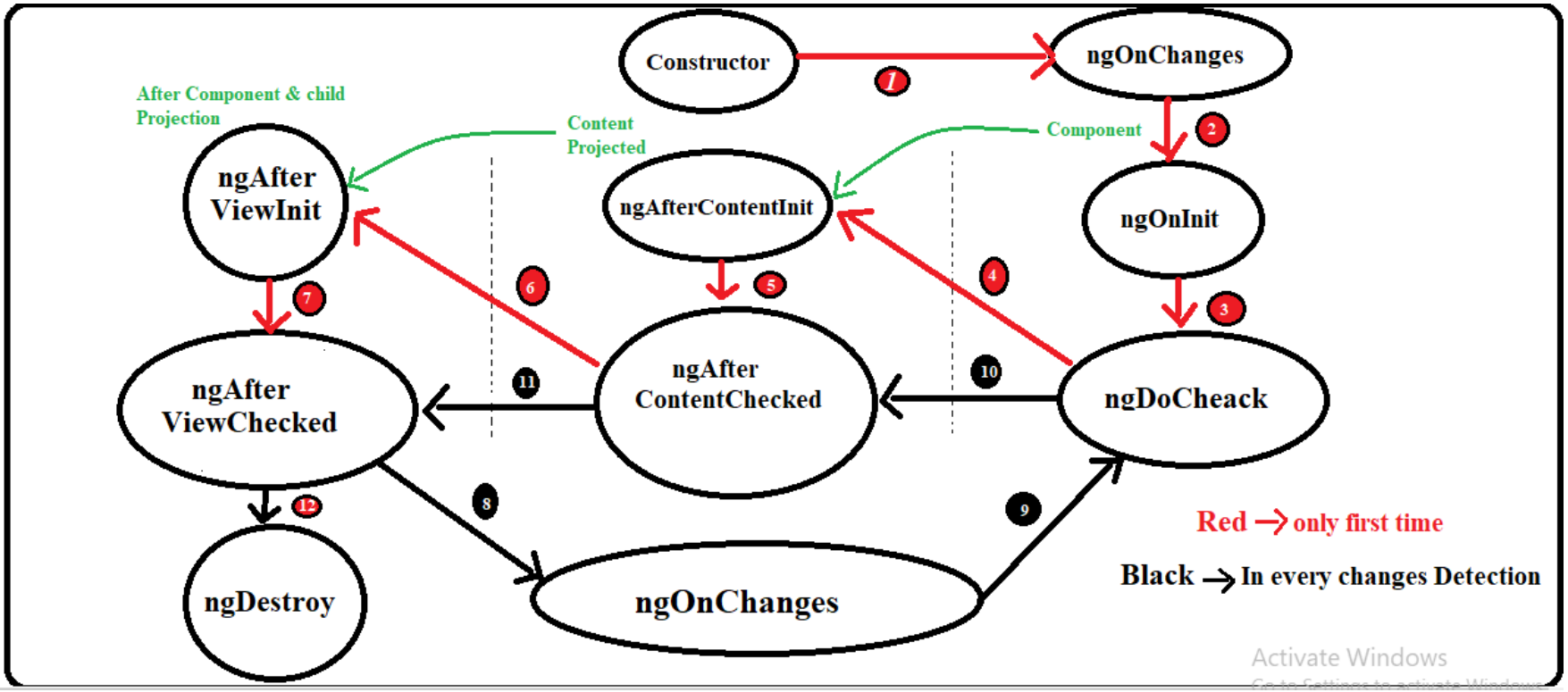
When the Component with Child Component is created

1. OnChanges
2. OnInit
3. DoCheck
4. AfterContentInit
5. AfterContentChecked
6. Child Component -> OnChanges
7. Child Component -> OnInit
8. Child Component -> DoCheck
9. Child Component -> AfterContentInit
10. Child Component -> AfterContentChecked
11. Child Component -> AfterViewInit
12. Child Component -> AfterViewChecked
13. AfterViewInit
14. AfterViewChecked

After The Component is Created

1. OnChanges
2. DoCheck
3. AfterContentChecked
4. AfterViewChecked

The OnChanges hook fires only if an input property is defined in the component and it changes. Otherwise, it will never fire.


Constructor
ngOnChanges
1
2
After Component & child Projection
Content Projected
Component
ngAfter ViewInit
ngAfterContentInit
ngOnInit
7
6
5
4
3
ngAfter ViewChecked
ngAfter ContentChecked
ngDoCheack
11
10
12
8
9
ngDestroy
ngOnChanges
Red → only first time
Black → In every changes Detection
Activate Windows