CSM
breaking thought barriers

Angular Training

Session -13

Outlines

- Decorator
- Angular Component
- Template VS TemplateUrl
- Nested Components
- Styling Components
- Databinding

CSM
breaking thought barriers

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.

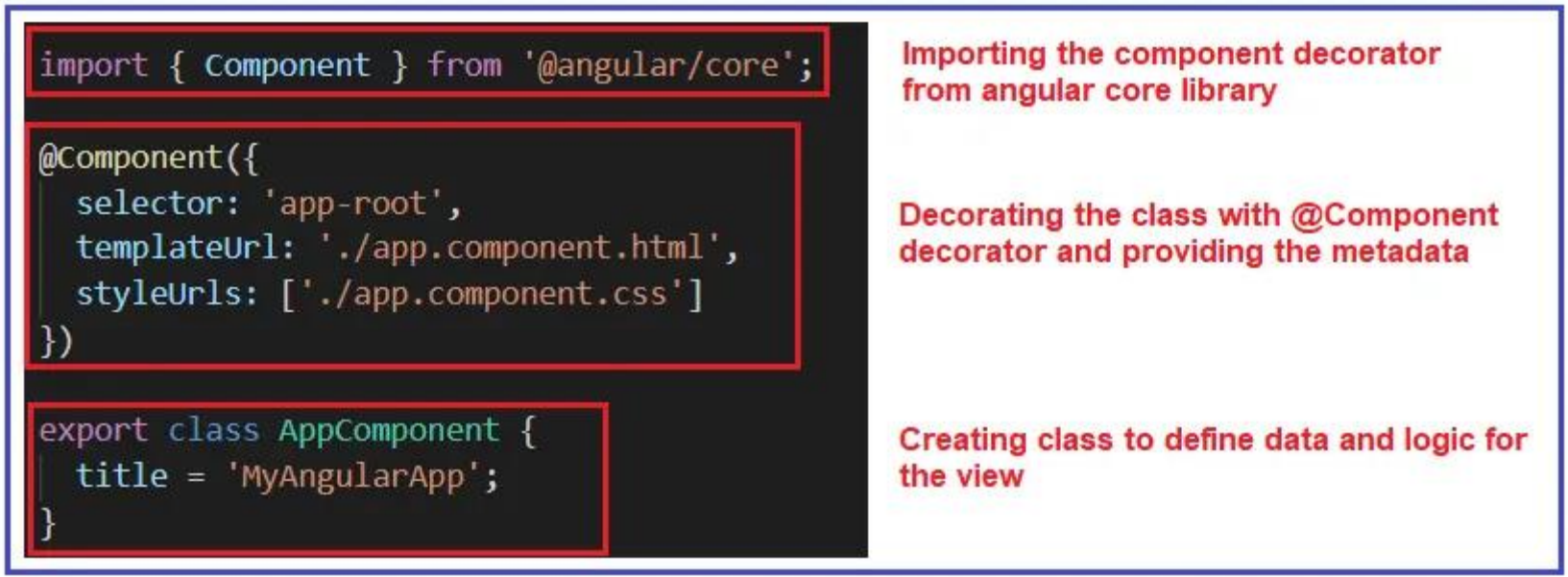
Class:

- ✓ 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?

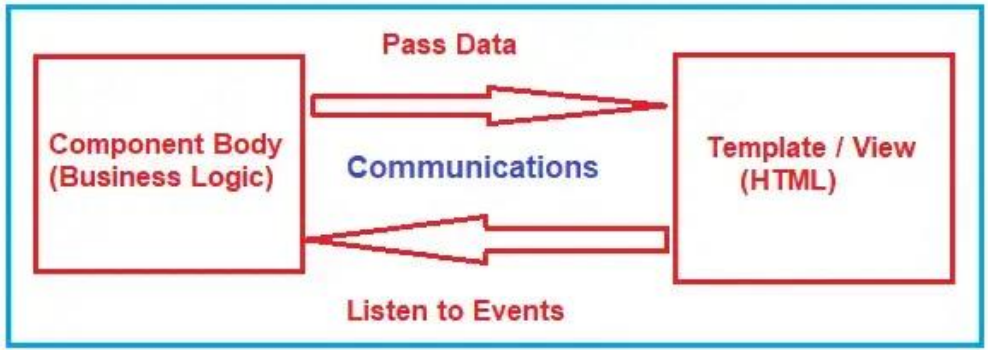
Whenever you want to develop any data-driven web application, then as a developer you need to keep the focus on two important things i.e. Data and the UI (User Interface) and it is more important for you to find an efficient way to bind them (Data and UI) together.

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.

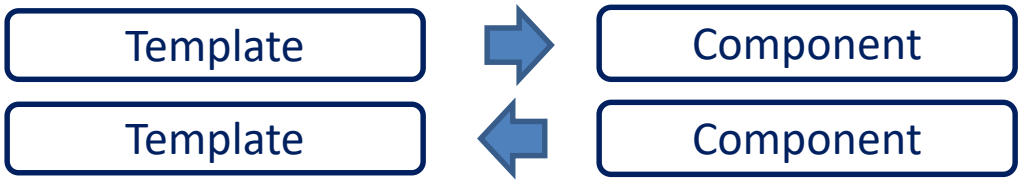
Data Binding is a process that creates a connection to communicate and synchronize between the user interface and the data. In order words, we can say that Data Binding means to interact with the data and view. So, the interaction between the templates (View) and the business logic is called data binding.



Types of Data Binding in Angular:

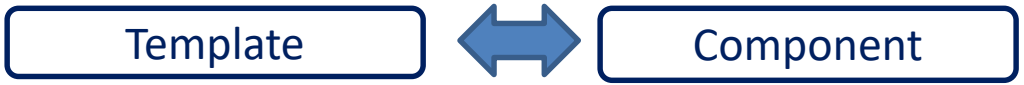
One-way Data Binding

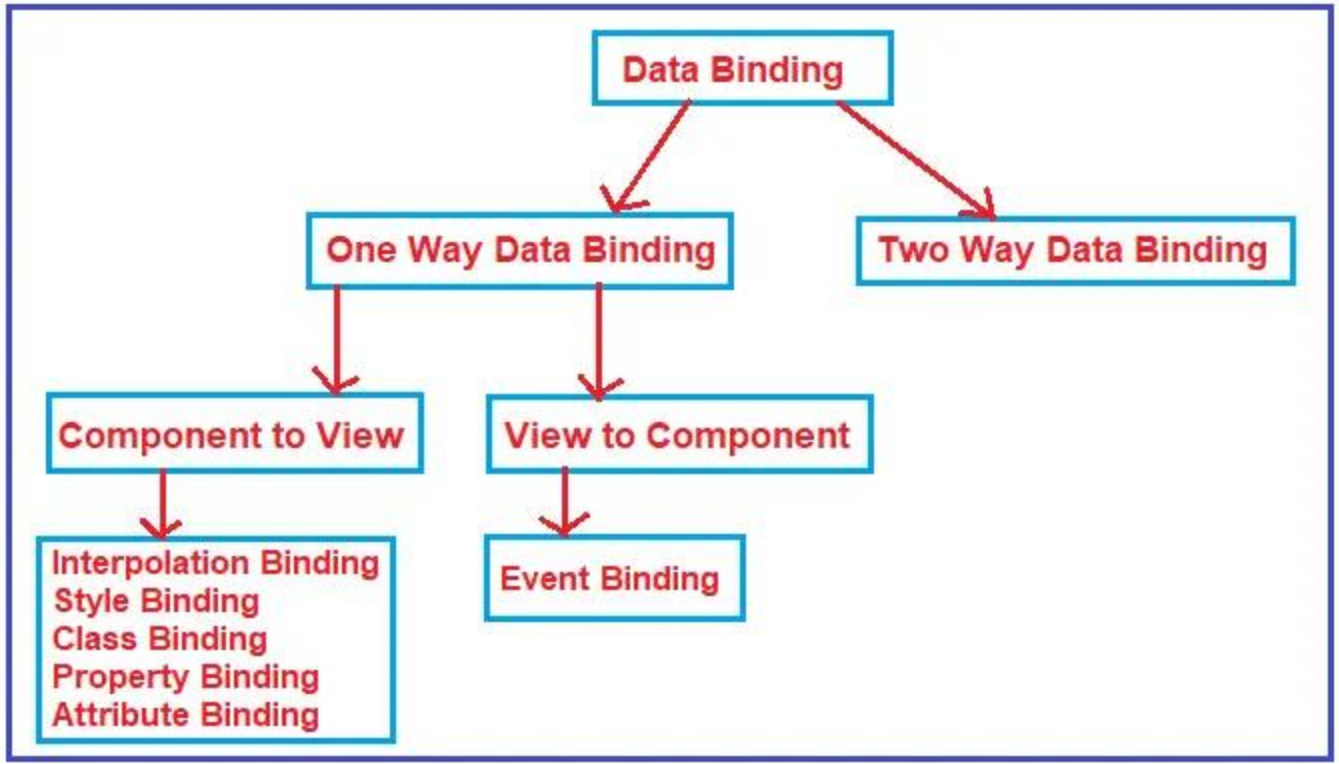
Where a change in the state affects the view (i.e. From Component to View Template) or change in the view affects the state (From View Template to Component).



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:

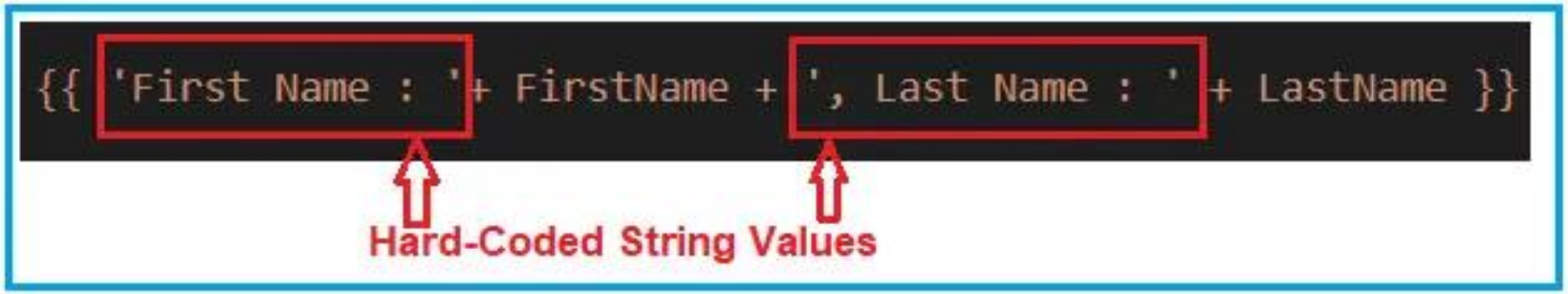
- ❑ Interpolation
- ❑ Property Binding
- ❑ Attribute Binding
- ❑ Class Binding
- ❑ Style Binding
- ❑ Event Binding
- ❑ Two-way binding

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.

- The Interpolation in Angular allows you to place the component property name in the view template, enclosed in double curly braces i.e. {{propertyName}}. So, the Angular Interpolation is a technique that allows the user to bind a value to a UI element.

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() }}
```

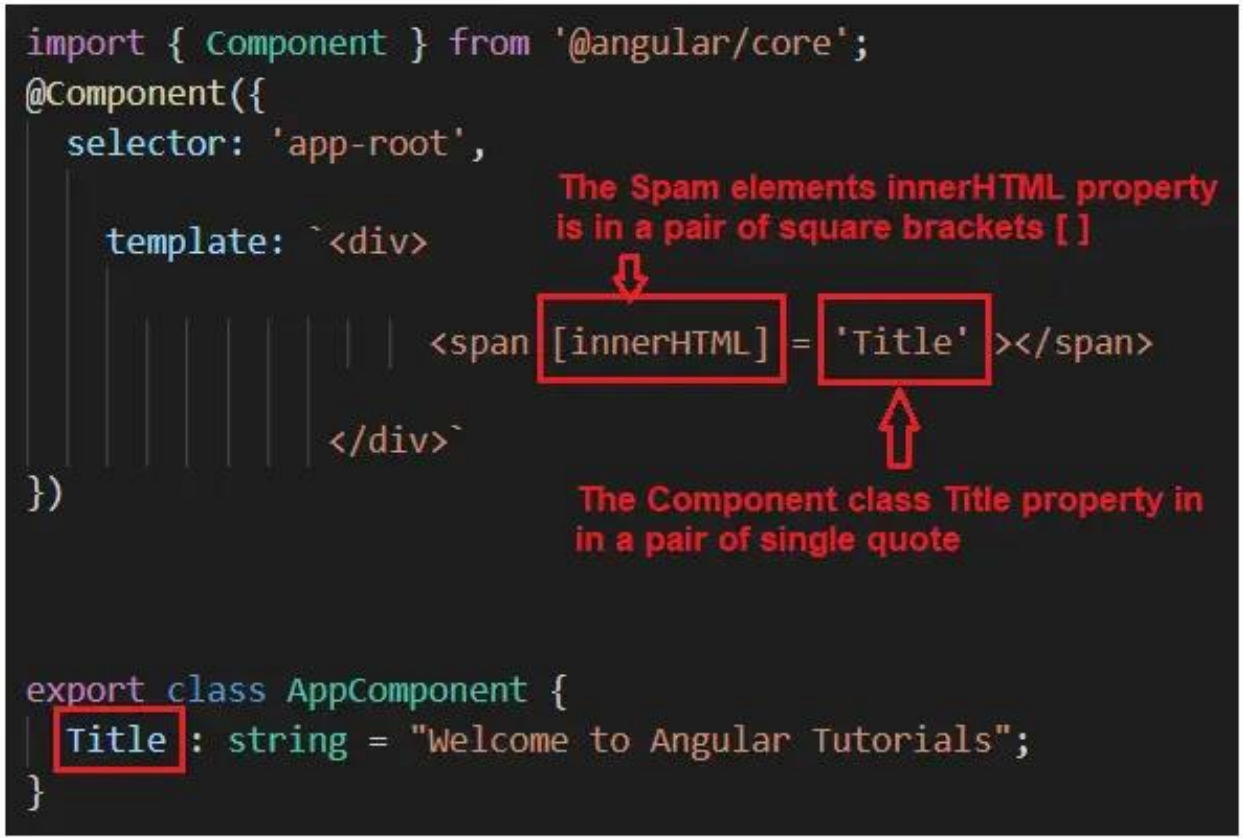
Displaying Images using Angular Interpolation:

Angular Property Binding

The Property Binding in Angular Application is used to bind the values of component or model properties to the HTML element. Depending on the values, it will change the existing behavior of the HTML element

The syntax : **[property] = ‘expression’**

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>";
}
```

HTML Attribute VS DOM Property

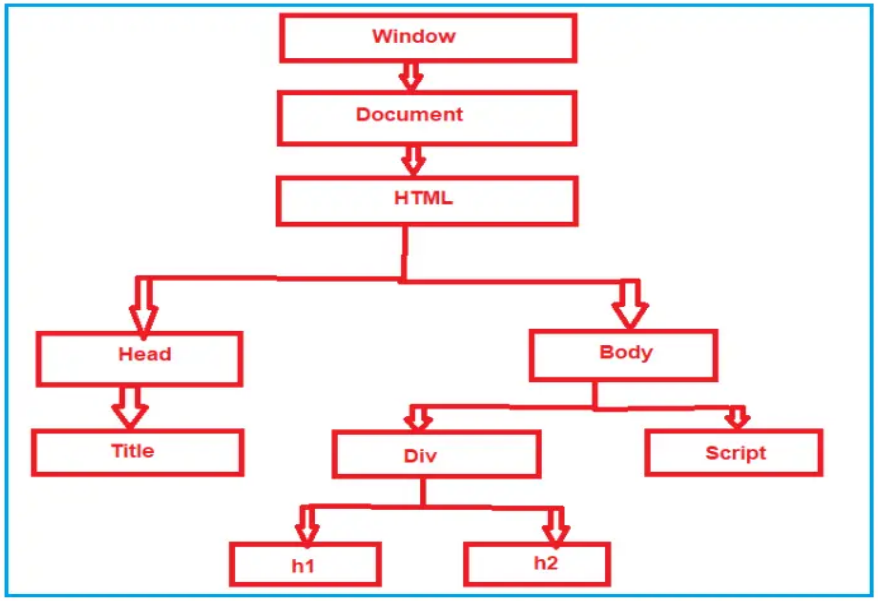
What is DOM?

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



We can say that the 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 include 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]='Apply BoldClass'>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>
```

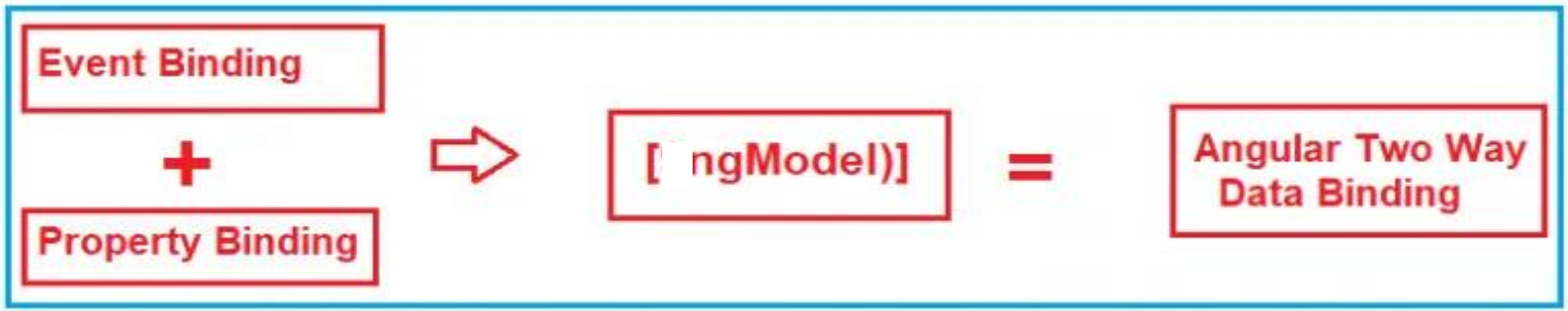
With event binding, you can also use the on- prefix alternative as shown below .This is known as the canonical form.

```
<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 1’ (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 [value]='Name' (input) = 'Name = $any($event.target).value
'>
            <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]