Data Binding

- Binding properties and Interpolation
- One-way Binding / Property Binding
- Event Binding
- Two-way Binding
- Two-way binding with NgModel
- Attribute Binding

Binding Properties

Angular provides many kinds of data binding, and we'll discuss each of them in this chapter. First we'll take a high-level view of Angular data binding and its syntax.

We can group all bindings into three categories by the direction in which data flows. Each category has its distinctive syntax:

Data direction	Syntax	Binding type
One-way from data source to view target	{{expression}} [target] = "expression" bind-target = "expression"	Interpolation Property Attribute Class Style
One-way from view target to data source	(target) = "statement" on-target = "statement"	Event
Two-way	[(target)] = "expression" bindon-target = "expression"	Two-way

Note: Binding types other than interpolation have a **target name** to the left of the equal sign, either surrounded by punctuation ([], ()) or preceded by a prefix (bind-, on-, bindon-).

Attributes vs DOM

A DOM tree is constructed from an HTML document. All elements take the form of nodes of the tree and

- A few HTML attributes have 1:1 mapping to properties. id is one example.
- Some HTML attributes don't have corresponding properties. colspan is one example.
- Some DOM properties don't have corresponding attributes. textContent is one example.
- Many HTML attributes appear to map to properties ... but not in the way we might think!

Attributes ***initialize*** DOM properties and then they are done.

Property values can change; attribute values can't.

eg: value attribute of <input> element.

The HTML attribute and the DOM property are not the same thing, even when they have the same name.

eg: disabled attribute and disabled property in <input> element.

Very Important:

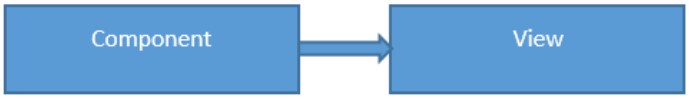
Template binding works with ***properties*** and ***events***, not ***attributes***.

Following are the Binding Types:

1. Property binding
2. Event Binding
3. Two-way Binding
4. Attribute Binding
5. Class Binding
6. Style Binding

One-Way Binding / Property Binding

- We write a template **property binding** when we want to set a property of a view element to the value of a template expression.
- People often describe property binding as *one-way data binding* because it flows a value in one direction, from a component's data property into a target element property.



- We **cannot use** property binding to pull values *out* of the target element. We can't bind to a property of the target element to read it. We can only set it.
- We can do one-way binding in four ways:
    a) Using double curly braces **{{expression}}** interpolation
    b) By wrapping the element property with square braces **[prop]**
    c) By adding **bind-** before the element property ex: **bind-value, bind-src**
    d) Using ngModel **[ngModel],** where ngModel is an angular directive.

```
@Component({
 selector: 'my-app',
 template: `
  <form>
    <input type="text" value="{{personName}}"/>
```

```
      <input type="text" [value]="personName"/>
      <input type="text" bind-value="personName"/>
      <input type="button" [disabled]="!IsModified" value="Click me"/>
      <input type="button" [disabled]="!IsModified" value="Click me"/>
  <input type="button" [hidden]="isHidden" value="Click me"/>
   </form>
`
})

export class AppComponent
{
   personName = "Sandeep Soni";
   isModified: boolean = false;
   isHidden: boolean = false;
}
```

Note that **interpolation** is a convenient alternative for **property binding** in many cases. In fact, Angular translates those interpolations into the corresponding property bindings before rendering the view.

There is no technical reason to prefer one form to the other. We lean toward readability, which tends to favor interpolation.

Content Security:

Fortunately, Angular data binding is on alert for dangerous HTML. It *sanitizes* the values before displaying them. It **will not** allow HTML with script tags to leak into the browser, neither with interpolation nor property binding.

```
evilTitle = 'Template <script>alert("evil never sleeps")</script> Syntax';


<p><span>"{{evilTitle}}" is the <i>interpolated</i> evil text.</span></p>
<p>"<span [innerHTML]="evilTitle"></span>" is the <i>property bound</i> evil text.</p>
```

Output:

"Template <script>alert("evil never sleeps")</script>Syntax" is the *interpolated* evil title.

"Template Syntax" is the *property bound* evil title.

Event Binding

Actions which result in a flow of data **from an element to a component**

The only way to know about a user action is to listen for certain events such as **keystrokes, mouse movements, clicks, and touches.**

Event Binding is also a **one-way binding** where it is exactly **opposite to the Property Binding** where it sends information form view to component class based on the user actions.

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

```
<button (click)="onSave()">Save</button>
```

Some people prefer the on- prefix alternative, known as the **canonical form**:

```
<button on-click="onSave()">On Save</button>
```

Here are events which are mostly used:

- Click
- Dblclick
- Mouseenter
- Mouseup
- Mousedown
- Keydown
- Keypress
- Keyup
- Cut
- Copy
- Paste
- Focus
- Blur
- Submit
- Scroll
- Drag
- Dragover
- Drop

***$event* and event handling statements**

In an event binding, Angular sets up an event handler for the target event.

When the event is raised, the handler executes the template statement. The template statement typically involves a receiver, which performs an action in response to the event, such as storing a value from the HTML control into a model.

Example:

```
@Component({
  selector: 'my-app',
  template: `
   <p>Events</p>
   <button type="button" (click) ="myEvent($event)">Click</button><br/>
   <button type="button" on-click="myEvent($event)">On-Click</button>
   <button type="button" (dblclick)="myEvent($event)">Double Click</button>
   <button type="button" (mouseover)="myEvent($event)">Mouse over</button>

   <input (keyup)="onKey($event)">
   <p>{{values}}</p>

`,
```

```
})
```

```
export class AppComponent {
  values = '';
  myEvent(event: any) {
    this.values = event.type + " event raised";
  }

  onKey(event: any) {
    this.values += event.key + ' | '; // event.target.value + ' | ';
  }
}
```

Note:

The properties of an $event object vary depending on the type of DOM event.

target property of $event references to the element that raised the event.

Template Reference Variable

These variables provide direct access to an element from within the template.

To declare a template reference variable, precede an identifier with a hash (or pound) character (#).

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

@Component({
  selector: 'my-app',
  template: `
    <input #txt1 (keyup)="0">
    <p>{{txt1.value}}</p>
`})

export class AppComponent { }
```

Note: **This won't work at all unless you bind to an event**.

Angular updates the bindings (and therefore the screen) only if the app does something in response to asynchronous events, such as keystrokes. This example code binds the keyup event to the number 0, the shortest

template statement possible. While the statement does nothing useful, it satisfies Angular's requirement so that Angular will update the screen.

It's easier to get to the input box with the template reference variable than to go through the $event object. Here's a rewrite of the previous keyup example that uses a template reference variable to get the user's input.

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

@Component({
  selector: 'my-app',
  template: `
    <input #txt1 (keyup)="onKey(txt1.value)">
    <p>{{values}}</p>
`})

export class AppComponent {
  values = '';

  onKey(value: string) { // without type info
    this.values += value + ' | ';
  }
}
```

Key event filtering (with key.enter)

The (keyup) event handler hears *every keystroke*. Sometimes only the *Enter* key matters, because it signals that the user has finished typing. One way to reduce the noise would be to examine every $event.keyCode and take action only when the key is *Enter*.

There's an easier way: bind to Angular's keyup.enter pseudo-event. Then Angular calls the event handler only when the user presses *Enter*.

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

@Component({
  selector: 'my-app',
  template: `
```

```
    <input #txt1 (keyup.enter)="onKeyEnter(txt1.value)">
    <p>{{values}}</p>
`})

export class AppComponent {
  values = '';

  onKeyEnter(value: string) { // without type info
    this.values += value + ' | ';
  }
}
```

Mouse Event Example

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

@Component({
  selector: 'my-app',
  template: `
    <img src="app/smiley.jpg" #img
      (mouseover)=onMouseOver(img)
      (mouseout)=onMouseOut(img)/>
`})

export class AppComponent {
  onMouseOver(img: any) {
    img.src = "app/smiley.jpg";
  }

  onMouseOut(img: any) {
    img.src = "app/capture.jpg";
  }
}
```

Two-way binding

- We often want to both display a data property and update that property when the user makes changes.

- On the element side that takes a combination of setting a specific element property and listening for an element change event.

Two-way binding using (input) event:

```
<p>Two way binding using (input)</p>
<p>Hello {{myname}}</p>
<input type="text"  [value]="myname" (input)="myname = $event.target.value" />
```

- **(input)=”expression”** - Is a declarative way of binding an expression to the input element’s **input event.**
- **myname = $event.target.value** - The expression that gets executed when the input event is fired
- **$event** - Is an expression exposed in event bindings by Angular, which has the value of the event’s payload

Output:

Two way binding using (input)

Hello Sandeep Soni

Sandeep Soni

Two-way binding with NgModel

- Angular offers a special ***two-way data binding*** syntax for this purpose, **[(x)]**.
- The [(x)] syntax combines the brackets of *property binding*, [x], with the parentheses of *event binding*, (x).
- The [(x)] syntax is easy to demonstrate when the element has a settable property called x and a corresponding event named xChange
- The **ngModel** directive is part of a built-in Angular module called "**FormsModule**". So, we must import this module in to the template module before using the **ngModel** directive.

When developing data entry forms, we often want to both display a data property and update that property when the user makes changes. Two-way data binding with the NgModel directive makes that easy.

The **ngModel** data property sets the element's value property and the **ngModelChange** event property listens for changes to the element's value.

app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
```

```
import { FormsModule } from '@angular/forms'
import { AppComponent }  from './app.component';

@NgModule({
 imports:     [ BrowserModule, FormsModule ],
 declarations: [ AppComponent ],
 bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

app.component.ts

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

@Component({
   selector: 'my-app',
   template: `
    First Name: <input [(ngModel)]="personFirstName">
    <br/>
    Last Name: <input  [ngModel]="personLastName"
              (ngModelChange)="personLastName=$event">
    <hr/>
    Hello {{personFirstName}} {{personLastName}}
`,
})
export class AppComponent {
   personFirstName: string = 'Sandeep';
   personLastName: string = 'Soni';
}
```

Note: The details are specific to each kind of element and therefore the **NgModel** directive only works for specific form elements, such as the input text box, that are supported by a **ControlValueAccessor** (Interface which is a bridge between a control and a native element).

Also note that the [(ngModel)] syntax can only *set* a data-bound property. If we need to do something more or something different, we need to write the expanded form ourselves.

```
<input [ngModel]="emp. Name"
```

```
          (ngModelChange)="setUpperCaseName($event)">


setUpperCaseName(event: any)
{
  this.personLastName = event.toUpperCase();
}
```

Using @Input and @Output for Custom Events

@Input and **@Output** are imported from **@angular/core** library, where

- **@Input** decorator is used to define an input property to achieve **Component Property Binding**
- **@input:** This is one-way communication from **parent to child** component.

- **@Output** decorator is used to define output property to achieve **Custom Event Binding.**
- This is also a one-way communication **child to parent**.

- We can define its aliases as **@Input(alias)** and **@Output(alias)**

Component properties should be decorated using **@Input,** and it can be annotated at any type of properties like **string, number, array, user defined classes.**

Example 1:

sizer-component.ts

```
import { Component, EventEmitter, Input, Output } from '@angular/core';
//Child Component
@Component({
  selector: 'my-sizer',
  template: `
 <div>
  <label [style.font-size.px]="size">FontSize: {{size}}px</label>
  <button (click)="dec()" title="smaller">-</button>
  <button (click)="inc()" title="bigger">+</button>
 </div>`
})
export class SizerComponent {
  @Input() size: number =10 ;
  @Output() sizeChange = new EventEmitter<number>();
```

```
  dec() { this.resize(-1); }
  inc() { this.resize(+1); }

  resize(delta: number) {
    this.size = Math.min(40, Math.max(8, +this.size + delta));
    this.sizeChange.emit(this.size);
  }
}
```

App.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent }  from './app.component';
import { SizerComponent } from "./sizer.component"

@NgModule({
 imports:      [ BrowserModule,FormsModule ],
 declarations: [ AppComponent, SizerComponent ],
 bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

app.component.ts

```
import { Component } from '@angular/core';
import { EmployeeType } from "./employee"
import { Employee } from "./employee"
import { SizerComponent } from "./sizer.component"

//Parent Component
@Component({
 selector: 'my-app',
 template: `
   <my-sizer [(size)]="fontSizePx"></my-sizer>
   <div [style.font-size.px]="fontSizePx">Resizable Text</div>
`
```

```
})


export class AppComponent
{
   fontSizePx = "20";
}
```

Example 2:

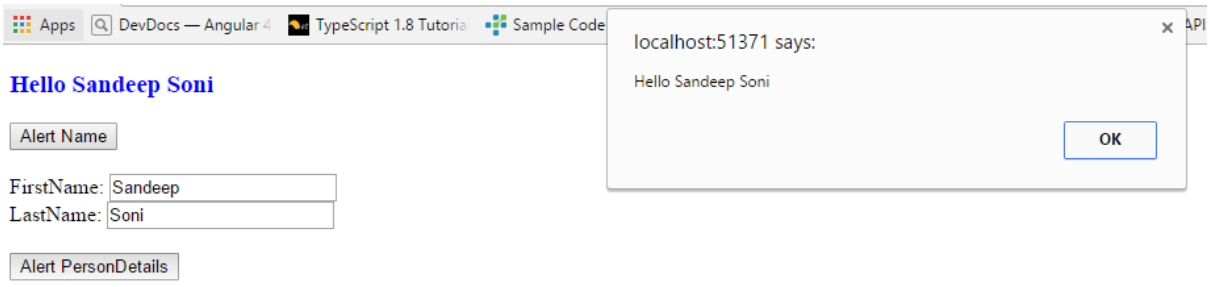
- Creating a Person class with **FName** and **LName** properties
- Creating component which can listen the events from the Student Class. Here **(getName)** and **(addPerson**) are the events that are created using **EventEmitter** in Student Class

File: **app.component.ts**

```
import { Component, Input, Output, EventEmitter, } from '@angular/core';
class Person {
    constructor(public FName?: string, public LName?: string) { }
}
//Parent Component
@Component({
    selector: 'my-app',
    template: `
    <Student name="Sandeep Soni"
            (evtName)=evtNameHandler($event)
            (evtPerson)=evtPersonHandler($event)>
    </Student>`
})
export class AppComponent {
    evtNameHandler(studName: string) {
        alert("Hello " + studName);
    }
    evtPersonHandler(person: Person) {
        alert("Hello " + person.FName + " " + person.LName);
    }
}

//Child Component: Student
@Component({
    selector: 'Student',
    template: `
      <h3>Hello {{name}}</h3>
```

```
      <button (click)=RaiseNameEvent(name)>Alert Name</button>
      <hr/>
      FirstName: <input type="text" (input)="person.FName=$event.target.value"/><br/>
      LastName: <input type="text" (input)="person.LName=$event.target.value"/><br/><br/>
      <button (click)=RaisePersonEvent()>Alert Person Details</button>
`,
})

export class Student {
    @Input() name: any;
    @Output("evtName") myNameEvent: EventEmitter<string> = new EventEmitter<string>();
    RaiseNameEvent(studName: string) {
        this.myNameEvent.emit(this.name);
    }

    @Output() evtPerson: EventEmitter<Person> = new EventEmitter<Person>();
    person = new Person();
    RaisePersonEvent() {
        this.evtPerson.emit(this.person);
    }
}
```

Output: Click on Alert Name gives you Hello Sandeep Soni, Alert Person Details gives the concatenation of FirstName and LastName



Attribute Binding

We have stressed throughout this chapter that setting an element property with a property binding is always preferred to setting the attribute with a string. Why does Angular offer attribute binding?

We must use attribute binding when there is no element property to bind.

Consider the **ARIA, SVG**, and table span attributes. They are pure attributes. They do not correspond to element properties, and they do not set element properties. There are no property targets to bind to.

We become painfully aware of this fact when we try to write something like this:

```
<tr><td colspan="{{1 + 1}}">One-Two</td></tr>
```

Error: Template parse errors: Can't bind to 'colspan' since it isn't a known native property

As the message says, the <td> element does not have a **colspan property**. It has the **"colspan" *attribute***, but interpolation and property binding can set only *properties*, not attributes.

Attribute binding syntax resembles property binding. Instead of an element property between brackets, we start with the prefix **attr**, followed by a dot (.) and the name of the attribute. We then set the attribute value, using an expression that resolves to a string.

```
<table border=1>
<!--  expression calculates colspan=2 -->
  <tr><td [attr.colspan]="1 + 1">One-Two</td></tr>

  <tr><td>Three</td><td>Four</td></tr>
</table>
```

Note: We will discuss Class Binding and Style Binding in Style Sheets chapter.