HOA SEN UNIVERSITY

www.hoasen.edu.vn

Angular – Template Syntax

Phạm Thị Kim Ngôn

Ngon.phamthikim@hoasen.edu.vn

Contents

- Signals
- Interpolation ({{...}})
- Property binding
- Class binding
- Style binding
- Event binding
- Control flow
- Pipe

Signals

- **What are signals?**
  - A signal is a wrapper around a value that notifies interested consumers when that value changes.
  - Signals can contain any value, from primitives to complex data structures.
- **Writable signals**

```
const count = signal(0);


// Signals are getter functions - calling them reads their value.
console.log('The count is: ' + count());
```

To change the value of a writable signal, either `.set()` or use the `.update()` operation to compute a new value from the previous one:

```
count.set(3);
```

```
// Increment the count by 1.
count.update(value => value + 1);
```

Computed - Signals

Computed signals

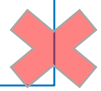
Computed signal are read-only signals that derive their value from other signals. You define computed signals using the computed function and specifying a derivation

```
const count: WritableSignal<number> = signal(0);
const doubleCount: Signal<number> = computed(() => count() * 2);
```

- Computed signals are both lazily evaluated and memorized
- Computed signals are not writable signals

You cannot directly assign values to a computed signal.

```
doubleCount.set(3);
```



because `doubleCount` is not a `WritableSignal`.

Computed - Signals

- Computed signal dependencies are dynamic

```
const showCount = signal(false);
const count = signal(0);
const conditionalCount = computed(() => {
  if (showCount()) {
    return `The count is ${count()}.`;
  } else {
    return 'Nothing to see here!';
  }
});
```

Effects - Signals

Signals are useful because they notify interested consumers when they change. An effect is an operation that runs whenever one or more signal values change.

```
effect(() => {
  console.log(`The current count is: ${count()}`);
});
```

Effects always execute **asynchronously**, during the change detection process.

Effects - Signals

Injection context

By default, you can **only create an effect()** within an **injection context** (where you have access to the inject function). The easiest way to satisfy this requirement is to **call effect** within **a component**, **directive**, or **service constructor**:

```
@Component({...})
export class EffectiveCounterComponent {
  readonly count = signal(0);
  constructor() {
    // Register a new effect.
    effect(() => {
      console.log(`The count is: ${this.count()}`);
    });
  }
}
```

Effects - Signals

Assign the effect to a field

```
@Component({...})
export class EffectiveCounterComponent {
  readonly count = signal(0);

  private loggingEffect = effect(() => {
    console.log(`The count is: ${this.count()}`);
  });
}
```

Effects - Signals

To create an effect **outside the constructor**, you can pass an Injector to effect via its options:

```
@Component({...})
export class EffectiveCounterComponent {
  readonly count = signal(0);
  private injector = inject(Injector);

  initializeLogging(): void {
    effect(() => {
      console.log(`The count is: ${this.count()}`);
    }, {injector: this.injector});
  }
}
```

Destroying effects - Signals

- When you create an effect, it is **automatically destroyed when its enclosing context is destroyed**.
- This means that effects created within components are destroyed when the component is destroyed. The same goes for effects within directives, services, etc.
- Effects return an EffectRef that you can use to destroy them **manually**, by calling the **.destroy() method**.

Interpolation ({{...}}) (1)

- The text between the braces is often the name of a component property.
- Angular replaces that name with the string value of the corresponding component property.

src/app/app.component.html

```
<p>My current hero is {{currentHero.name}}</p>
```

src/app/app.component.html

```
<h3>
  {{title}}
  <img src="{{heroImageUrl}}" style="height:30px">
</h3>
```

Interpolation ({{...}}) (2)

- The text between the braces is a template expression that Angular first evaluates and then converts to a string.

```
<!-- "The sum of 1 + 1 is 2" -->
<p>The sum of 1 + 1 is {{1 + 1}}</p>
```

```
<!-- "The sum of 1 + 1 is not 4" -->
<p>The sum of 1 + 1 is not {{1 + 1 + getVal()}}</p>
```

- A template expression produces a value. Angular executes the expression and assigns it to a property of a binding target; the target might be an HTML element, a component, or a directive.

[property]="expression"

Binding dynamic properties and attributes

- Native element properties

```
<!-- Bind the `disabled` property on the button element's DOM object -->
<button [disabled]="isFormValid()">Save</button>
```

- Component and directive properties

```
<!-- Bind the `value` property on the `MyListbox` component instance. -->
<my-listbox [value]="mySelection()" />
```

```
<!-- Bind to the `ngSrc` property of the `NgOptimizedImage` directive  -->
<img [ngSrc]="profilePhotoUrl()" alt="The current user's profile photo">
```

Binding dynamic properties and attributes

- Attributes

```
<ul [attr.role]="listRole()">
```

- Text interpolation in properties and attributes

```
<!-- Binds a value to the `alt` property of the image element's DOM object. -->
<img src="profile-photo.jpg" alt="Profile photo of {{ firstName() }}" >
```

```
<button attr.aria-label="Save changes to {{ objectType() }}">
```

HOA SEN UNIVERSITY

CSS class and style property bindings

- **CSS classes**

```
<!-- When `isExpanded` is truthy, add the `expanded` CSS class. -->
<ul [class.expanded]="isExpanded()">
```

```
@Component({
  template: `
    <ul [class]="listClasses"> ... </ul>
    <section [class]="sectionClasses()"> ... </section>
    <button [class]="buttonClasses()"> ... </button>
  `,
  ...
})
export class UserProfile {
  listClasses = 'full-width outlined';
  sectionClasses = signal(['expandable', 'elevated']);
  buttonClasses = signal({
    highlighted: true,
    embiggened: false,
  });
}
```

Output

```
<ul class="full-width outlined"> ... </ul>
<section class="expandable elevated"> ... </section>
<button class="highlighted"> ... </button>
```

CSS class and style property bindings

```
@Component({
  template: `<ul class="list" [class]="listType()" [class.expanded]="isExpanded()"> ...`,
  ...
})
export class Listbox {
  listType = signal('box');
  isExpanded = signal(true);
}
```

Output

```
<ul class="list box expanded">
```

CSS style properties

```
<!-- Set the CSS `display` property based on the `isExpanded` property. -->
<section [style.display]="isExpanded() ? 'block' : 'none'">
```

```
<section [style.height.px]="sectionHeightInPixels()">
```

CSS style properties

```
@Component({
  template: `
    <ul [style]="listStyles()"> ... </ul>
    <section [style]="sectionStyles()"> ... </section>
  `,
  ...
})
export class UserProfile {
  listStyles = signal('display: flex; padding: 8px');
  sectionStyles = signal({
    border: '1px solid black',
    'font-weight': 'bold',
  });
}
```

```
<ul style="display: flex; padding: 8px"> ... </ul>
<section style="border: 1px solid black; font-weight: bold"> ... </section>
```

Adding event listeners

- **Listening to native events**

```
@Component({
  template: `
    <input type="text" (keyup)="updateField($event)" />
  `,
  ...
})
export class AppComponent {
  updateField(event: KeyboardEvent): void {
    console.log(`The user pressed: ${event.key}`);
  }
}
```

Adding event listeners

- **Listening to native events**

```
@Component({
  template: `
    <input type="text" (keyup.enter)="updateField($event)" />
  `,
  ...
})
export class AppComponent{
  updateField(event: KeyboardEvent): void {
    console.log('The user pressed enter in the text field.');
  }
}
```

```
<!-- Matches shift and enter -->
<input type="text" (keyup.shift.enter)="updateField($event)" />
```

CONTROL FLOW

@if, @else-if and @else

```
@if (a > b) {
  <p>{{a}} is greater than {{b}}</p>
}
```

```
@if (a > b) {
  {{a}} is greater than {{b}}
} @else if (b > a) {
  {{a}} is less than {{b}}
} @else {
  {{a}} is equal to {{b}}
}
```

```
@if (user.profile.settings.startDate; as startDate) {
  {{ startDate }}
}
```

@for

```
@for (item of items; track item.id) {
  {{ item.name }}
}
```

@for

Variable	Meaning
$count	Number of items in a collection iterated over
$index	Index of the current row
$first	Whether the current row is the first row
$last	Whether the current row is the last row
$even	Whether the current row index is even
$odd	Whether the current row index is odd

```
@for (item of items; track item.id; let idx = $index, e = $even) {
  <p>Item #{{ idx }}: {{ item.name }}</p>
}
```

@for, @empty

```
@for (item of items; track item.name) {
  <li> {{ item.name }}</li>
} @empty {
  <li> There are no items. </li>
}
```

@switch

```
@switch (userPermissions) {
  @case ('admin') {
    <app-admin-dashboard />
  }
  @case ('reviewer') {
    <app-reviewer-dashboard />
  }
  @case ('editor') {
    <app-editor-dashboard />
  }
  @default {
    <app-viewer-dashboard />
  }
}
```

Pipes

- Pipes were earlier called filters in Angular1 and called pipes in Angular 2 and 4.
- The | character is used to transform data.

```
{{ firstName + lastName | uppercase }}
```

- It takes integers, strings, arrays, and date as input separated with | to be converted in the format as required and display the same in the browser.

Pipes - Example

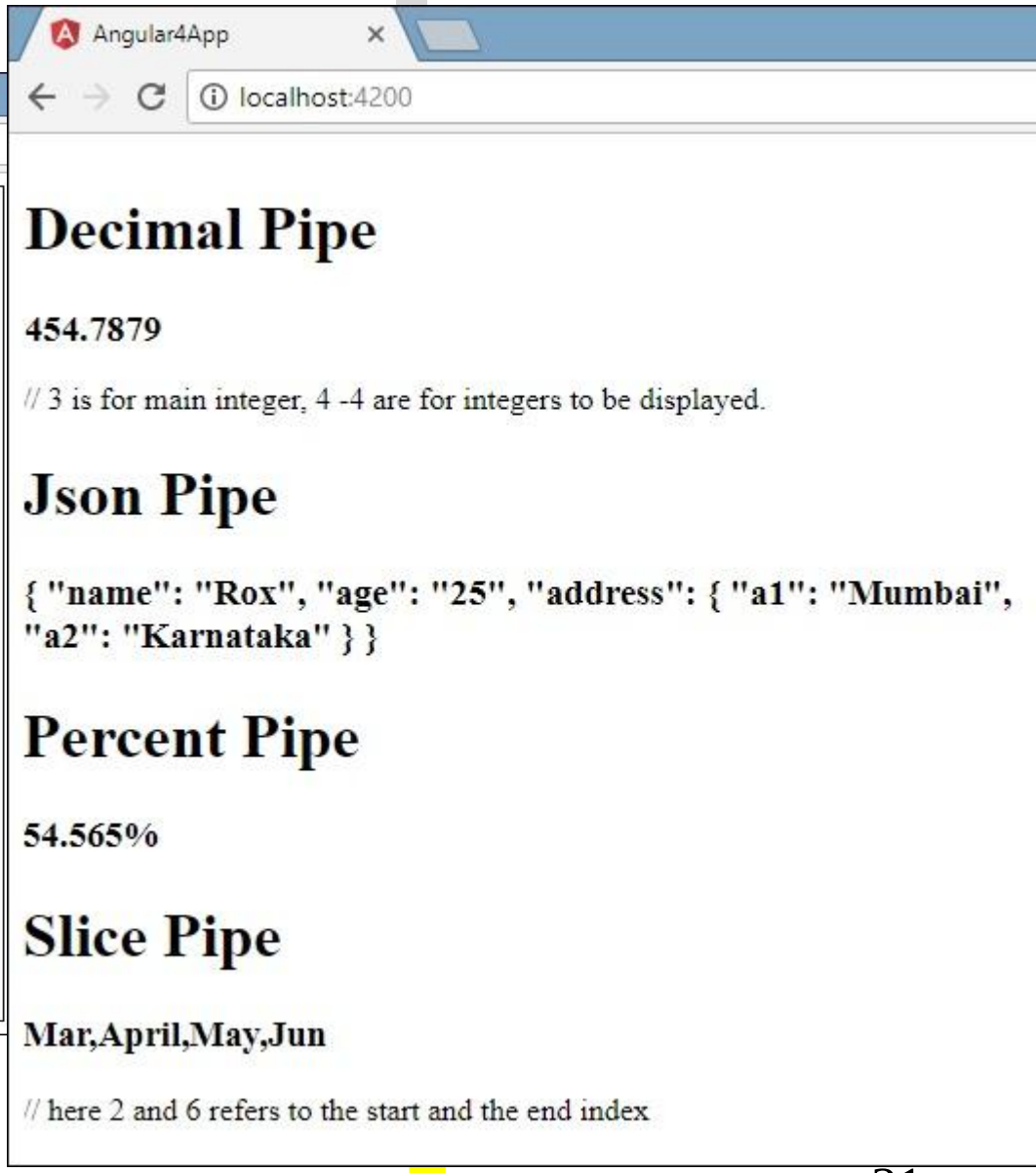
```
import { Component } from '@angular/core';
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})

export class AppComponent {
    title = 'Angular 4 Project!';
}
```

```
<b>{{title | uppercase}}</b><br/>
<b>{{title | lowercase}}</b>
```

Name	Description
AsyncPipe	Read the value from a Promise or an RxJS Observable.
CurrencyPipe	Transforms a number to a currency string, formatted according to locale rules.
DatePipe	Formats a Date value according to locale rules.
DecimalPipe	Transforms a number into a string with a decimal point, formatted according to locale rules.
I18nPluralPipe	Maps a value to a string that pluralizes the value according to locale rules.
I18nSelectPipe	Maps a key to a custom selector that returns a desired value.
JsonPipe	Transforms an object to a string representation via JSON.stringify, intended for debugging.
KeyValuePipe	Transforms Object or Map into an array of key value pairs.
LowerCasePipe	Transforms text to all lower case.
PercentPipe	Transforms a number to a percentage string, formatted according to locale rules.
SlicePipe	Creates a new Array or String containing a subset (slice) of the elements.
TitleCasePipe	Transforms text to title case.
UpperCasePipe	Transforms text to all upper case.

```
<div style = "width:40%;float:left;border:solid 1px black;">
<h1>Uppercase Pipe</h1>
<b>{{title | uppercase}}</b><br/>
<h1>Lowercase Pipe</h1>
<b>{{title | lowercase}}</b>
<h1>Currency Pipe</h1>
<b>{{6589.23 | currency:"USD"}}</b><br/>
<b>{{6589.23 | currency:"USD":true}}</b> //Boolean true is used to get
the sign of the currency.
<h1>Date pipe</h1>
<b>{{todaydate | date:'d/M/y'}}</b><br/>
<b>{{todaydate | date:'shortTime'}}</b>
<h1>Decimal Pipe</h1>
<b>{{ 454.78787814 | number: '3.4-4' }}</b> // 3 is for main integer, 4
-4 are for integers to be displayed.
</div>
<div style = "width:40%;float:left;border:solid 1px black;">
<h1>Json Pipe</h1>
<b>{{ jsonval | json }}</b>
<h1>Percent Pipe</h1>
<b>{{00.54565 | percent}}</b>
<h1>Slice Pipe</h1>
<b>{{months | slice:2:6}}</b>
</div>
```


Angular4App
localhost:4200
Uppercase Pipe
ANGULAR 4 PROJECT!
Lowercase Pipe
angular 4 project!
Currency Pipe
USD6,589.23
$6,589.23
//Boolean true is used to get the sign of the currency.
Date pipe
1/8/2017
10:48 PM


Angular4App
localhost:4200
Decimal Pipe
454.7879
// 3 is for main integer, 4 -4 are for integers to be displayed.
Json Pipe
{ "name": "Rox", "age": "25", "address": { "a1": "Mumbai", "a2": "Karnataka" } }
Percent Pipe
54.565%
Slice Pipe
Mar,April,May,Jun
// here 2 and 6 refers to the start and the end index

How to Create a Custom Pipe?

app.sqrt.ts

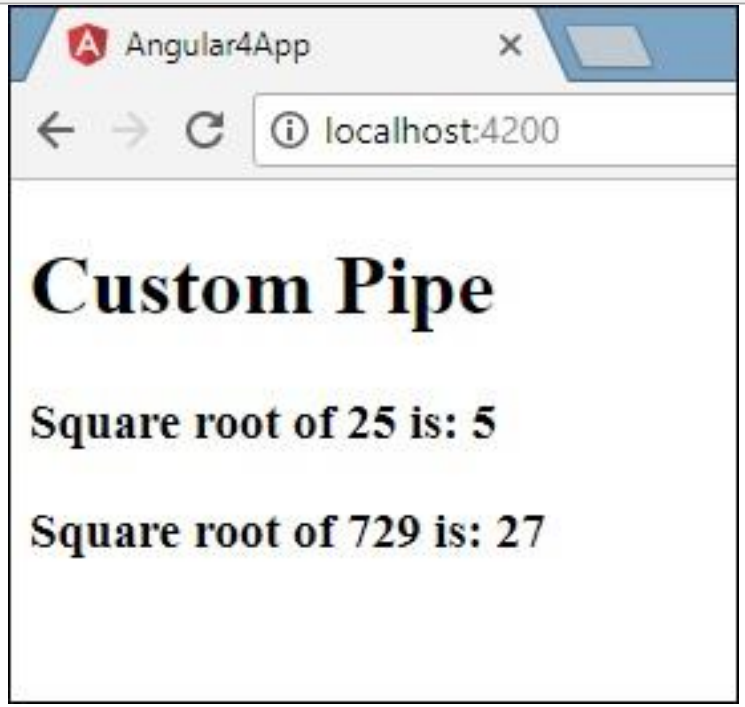
```
import {Pipe, PipeTransform} from '@angular/core';
@Pipe ({
        name : 'sqrt'
})
export class SqrtPipe implements PipeTransform {
    transform(val : number) : number {
        return Math.sqrt(val);
    }
}
```

app.module.ts

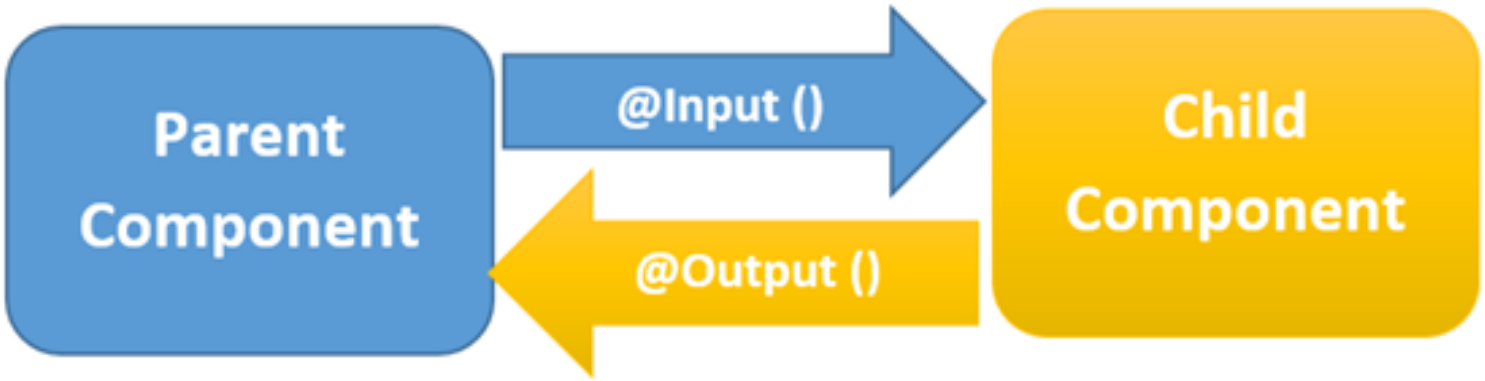
```
...
import { SqrtPipe } from './app.sqrt';
@NgModule({
    declarations: [
        SqrtPipe,
        AppComponent,
        NewCmpComponent,
        ChangeTextDirective
    ],
...
```

app.component.html

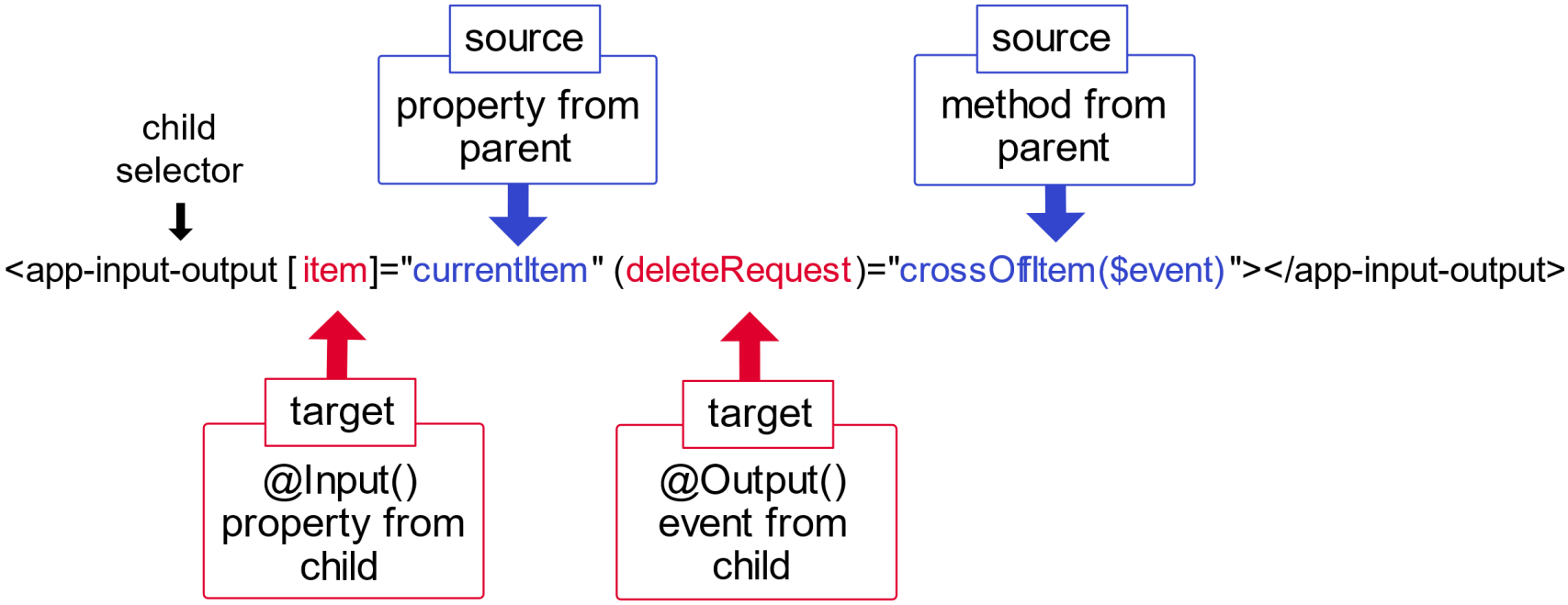
```
<h1>Custom Pipe</h1>
<b>Square root of 25 is: {{25 | sqrt}}</b>
<br/>
<b>Square root of 729 is: {{729 | sqrt}}</b>
```



Declaring Input and Output properties



Declaring Input and Output properties



Declaring Input and Output properties

src/app/hero-detail.component.ts

```
@Input()  hero: Hero;
@Output() deleteRequest = new EventEmitter<Hero>();
```

OR

```
@Component({
  inputs: ['hero'],
  outputs: ['deleteRequest'],
})
```

Input

Output

```
<hero-detail [hero]="currentHero" (deleteRequest)="deleteHero($event)">
```

Two-way binding ([(...)])

sizer.component.ts

```
import { Component, OnInit, EventEmitter, Input, Output } from
'@angular/core';
@Component({
    selector: 'sizer',
    templateUrl: './sizer.component.html',
    styleUrls: ['./sizer.component.css']
})
export class SizerComponent implements OnInit {
    constructor() { }

    ngOnInit( ) {
        this.size = 10;
    }
    @Input() size: number | string;
    @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);
    }
}
```

Two-way binding ([(...)])

sizer.component.html

```
<div>
    <button (click)="dec()" title="smaller">-</button>
    <button (click)="inc()" title="bigger">+</button>
    <label [style.font-size.px]="size">FontSize:
    {{size}}px</label>
</div>
```

app.component.html

```
<div id="two-way-1">
<h2>TWO - WAY BINDING 1 </h2>
<app-sizer [size]="fontSizePx"
(sizeChange)="fontSizePx=$event"></app-sizer>
<label>FontSize (px): <input [(ngModel)]="fontSizePx"></label>
</div>
```

Two-way binding ([(...)])

app.component.html

```
<h2>TWO - WAY BINDING 2 </h2>
<sizer [(size)]="fontSizePX"></sizer>
<div [style.font-size.px]="fontSizePx">Resizable Text</div>
<label>FontSize (px): <input [(ngModel)]="fontSizePx"></label>
```

Binding syntax

Data direction	Syntax	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

Binding targets

Type	Target	Examples

Property	Element property
	Component property
	Directive property
		`<img [src]="heroImageUrl">`
		`<hero-detail [hero]="currentHero"></hero-detail>`
		`<div [ngClass]="{special: isSpecial}"></div>`

Event	Element event
	Component event
	Directive event
		`<button (click)="onSave()">Save</button>`
		`<hero-detail (deleteRequest)="deleteHero()">`
		`</hero-detail>`
		`<div (myClick)="clicked=$event" clickable>click me</div>`

Two-way	Event and property	`<input [(ngModel)]="name">`
Attribute	Attribute (the exception)	`<button [attr.aria-label]="help">help</button>`
Class	class property	`<div [class.special]="isSpecial">Special</div>`
Style	style property	`<button [style.color]="isSpecial ? 'red' : 'green'">`

References