

Components Inputs/Outputs

CS569 – Web Application Development II

Maharishi International University

Department of Computer Science

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

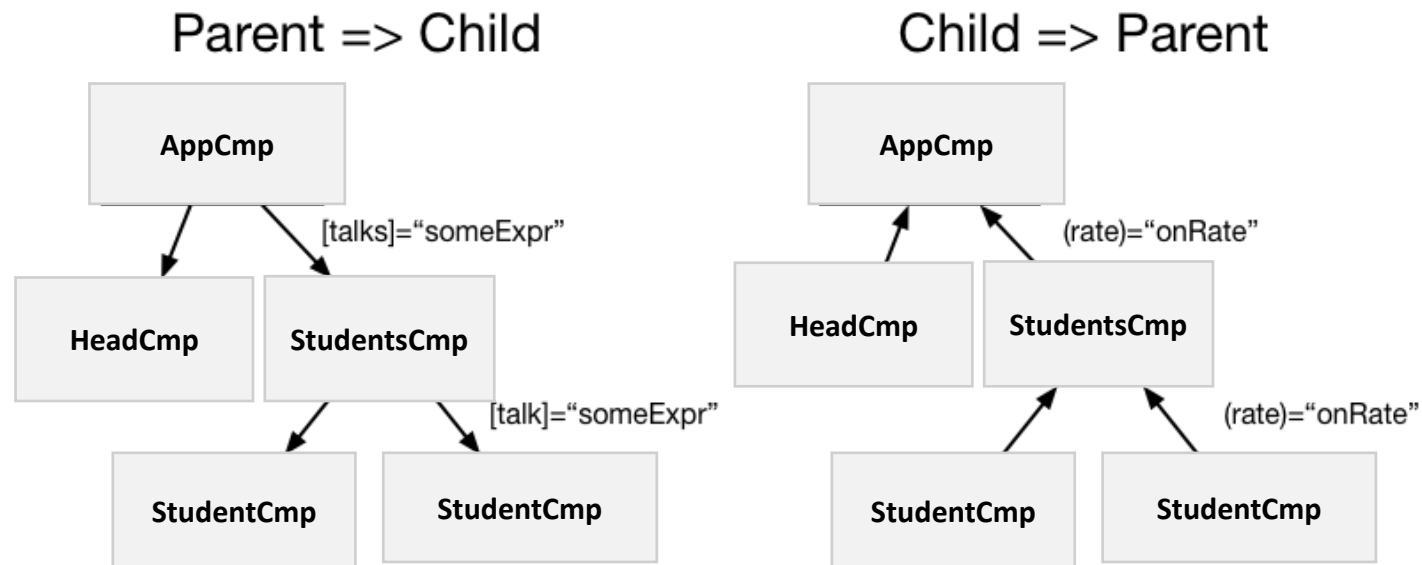


All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi International University.

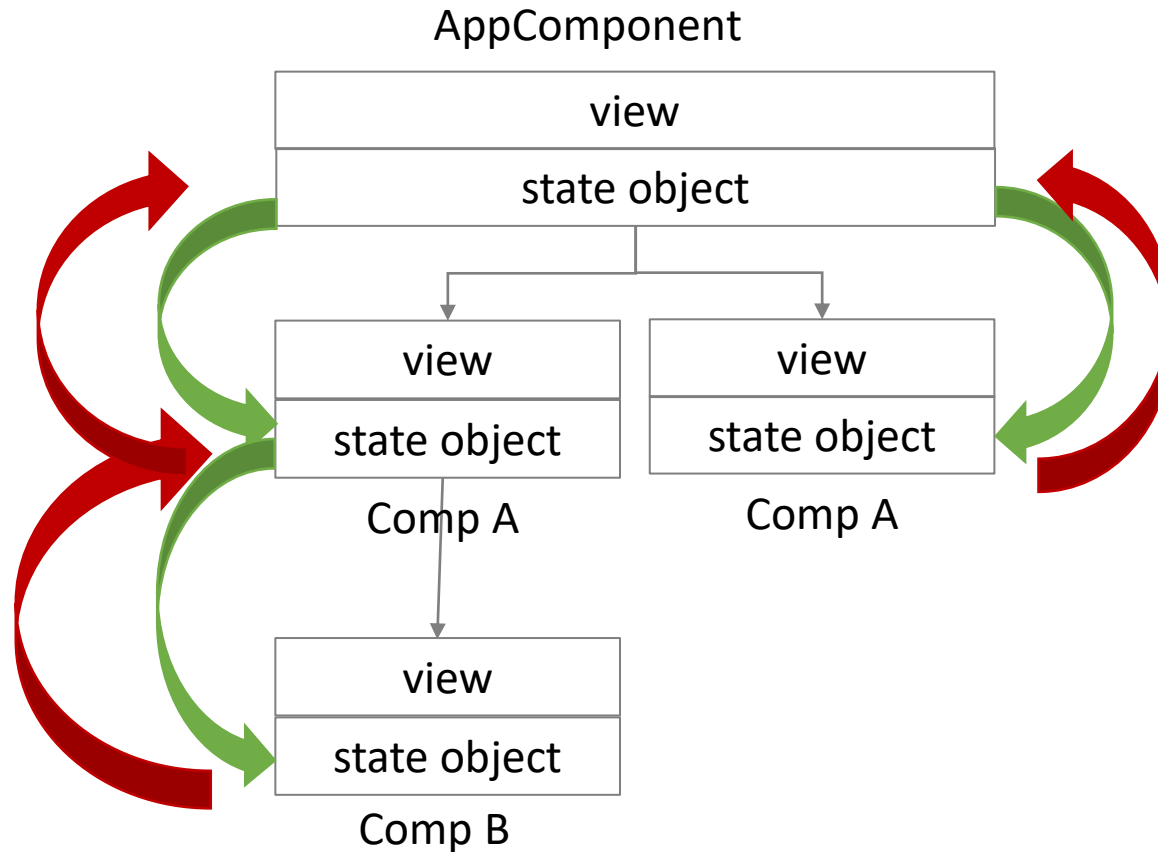
Input and Output Properties

A component has **input** and **output** properties, which can be defined in the component decorator or using class property decorators.

Data flows into a component via input properties. Data flows out of a component via output properties.



Communication: **Input** and **Output**



Communication between components: a parent component may pass data from their state object to their child state object through **inputs**, a child component may pass data to its parent state object through **outputs**.

Inputs and Outputs

Input and output properties are the public API of a component. You use them when you instantiate a component in your application.

```
<myComponent  
  [color]="colorValue" <!-- input -->  
  (onComponentSelected)="componentWasSelected($event)" > <!-- output -->  
</myComponent>
```

\$event is a special variable here that represents the thing being emitted.

[squareBrackets] pass inputs: You can set input properties using property bindings
(parens) handle outputs: You can subscribe to output properties using event bindings

Passing Input/Output

```
<input name="username" onkeyup="doSomething()" />
```

Using JS to pass input/output to
<input> Native Component

```
@Component({ template: `<input [name]="field" (keyup)="doSomething()" />` })
```

```
class Component {  
  field = 'username';  
  doSomething() {}  
}
```

Using Angular to pass input/output to
<input> Native Component

```
@Component({ template: `<weather [unit]="tempUnit" (onStorm)="doSomething()"></weather>` })
```

```
class Component {  
  tempUnit = 'F';  
  doSomething() {}  
}
```

Using Angular to pass input/output to **<weather>** Custom Component

```
class Weather {  
  @Input() unit;  
  @Output() onStorm = new EventEmitter();  
}
```

Native Components

Every native component, by default, has inputs and outputs:

For example: `<input />`

Native input properties: `value`, `class`, `id`, `type`.. etc

All attributes are input

Native output properties: `click`, `mouseover`, `input`, `keyup`.. etc

All events are output

We can simply interact with it:

```
<input [value]="prop" (keyup)="doSomething()" />
```

Component inputs

With the **inputs** property, we specify the parameters we expect our component to receive. Inputs takes an array of strings which specify the input keys.

When we specify that a Component takes an input, it is expected that the class will have an instance variable that will receive the value.

```
@Component({  
  selector: 'my-component',  
  inputs: ['name', 'age']  
})  
export class MyComponent {  
  name: string;  
  age: number;  
}
```



```
@Component({  
  selector: 'my-component'  
})  
export class MyComponent {  
  @Input() name: string;  
  @Input() age: number;  
}
```

```
<my-component [name]="myName" [age]="myAge"></my-component>
```


Passing data to inputs

```
@Component ({
    template: `
        <my-course [grade]="studentGrade" ></my-course>
        <my-course grade="{{studentGrade}}" ></my-course>
        <my-course grade="100" ></my-course>
    `,
})
export class ParentComponent {
    studentGrade: 100;
}
```

<ng-content>

```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'comp',
  template: ` <ul>
    <li>Message 1: {{message1}}</li>
    <li>Message 2: {{message2}}</li>
  </ul>
  <ng-content></ng-content> `
})
export class Comp1Component {
  @Input() message1;
  @Input() message2;
}
```

Property Binding: When we add an attribute in brackets like [foo] we're saying we want to pass an **expression value** to the input named foo on that component.

```
<comp message1="{{msg}}" [message2]="msg">Hey</comp>
```

Assume having msg="Hi"; in the parent component

Component outputs

When we want to send data out from a component, we use **output bindings**.

```
@Component({
  selector: 'my-component',
  outputs: ['onLectureEnds']
})
export class MyComponent {
  onLectureEnds = new EventEmitter();
}
```



```
@Component({
  selector: 'my-component'
})
export class MyComponent {
  @Output() onLectureEnds: new EventEmitter();
}
```

```
<my-component (onLectureEnds)="doSomething()"></my-component>
```

Native Element Output Example

```
@Component({
  selector: 'counter',
  template: `{{ value }}
             <button (click)="increase()">Increase</button> `
})
export class Counter {
  value: number;
  constructor() {
    this.value = 1;
  }

  increase() {
    this.value = this.value + 1;
    return false;
  }
}
```

tells the browser not to propagate the event upwards

Native Element Output Example

```
@Component({
  selector: 'counter',
  template: ` {{ value }}
               <button (click)="increase($event)">Increase</button> `
})
export class Counter {
  value: string;
  constructor() {
    this.value = 'no click';
  }

  increase(e) {
    this.value = e.target.innerHTML;
    return false;
  }
}
```

What would \$event refer to in this example?

Emitting Custom Events

Let's say we want to create a component that emits a custom event, just like native components events `"click"` or `"mousedown"`.

To create a custom output event we do three things:

1. Specify `outputs` property
2. Attach an **EventEmitter** to the output property
3. Emit an event from the **EventEmitter** , at the right time

An `EventEmitter` is simply an object that helps you implement the Observer Pattern. It's an object that can maintain a list of subscribers and publish events to them.

Custom Event Example

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

```
@Component({  
  selector: 'lecture',  
  outputs: ['onLunchBreak'],  
  template: `<button (click)="start()">Start Lunch Break</button>`  
})
```

1. specified outputs

```
class OutputComponent {  
  onLunchBreak: EventEmitter<string>;  
  constructor() {  
    this.onLunchBreak = new EventEmitter();  
  }  
  
  start(): void {  
    this.onLunchBreak.emit("Yes finally!! I'm hungry!");  
  }  
}
```

2. created an EventEmitter that we attached to the output property onLunchBreak

3. Emit an event when start() is called

Custom Event Example - Continued

If we wanted to use this output in a parent component we could do something like this:


```
@Component({
  selector: 'university',
  template: `<div>
    <lecture (onLunchBreak)="eatFood($event)"></lecture>
  </div> ` })
```

onLunchBreak comes from the outputs of OutputComponent



```
class UniversityComponent {
  eatFood(e: string) {
    console.log(`Message: ${e}`); // Yes finally!! I'm hungry!
  }
}
```

\$event contains the thing that
was emitted, in this case a string



Two-way Data-binding

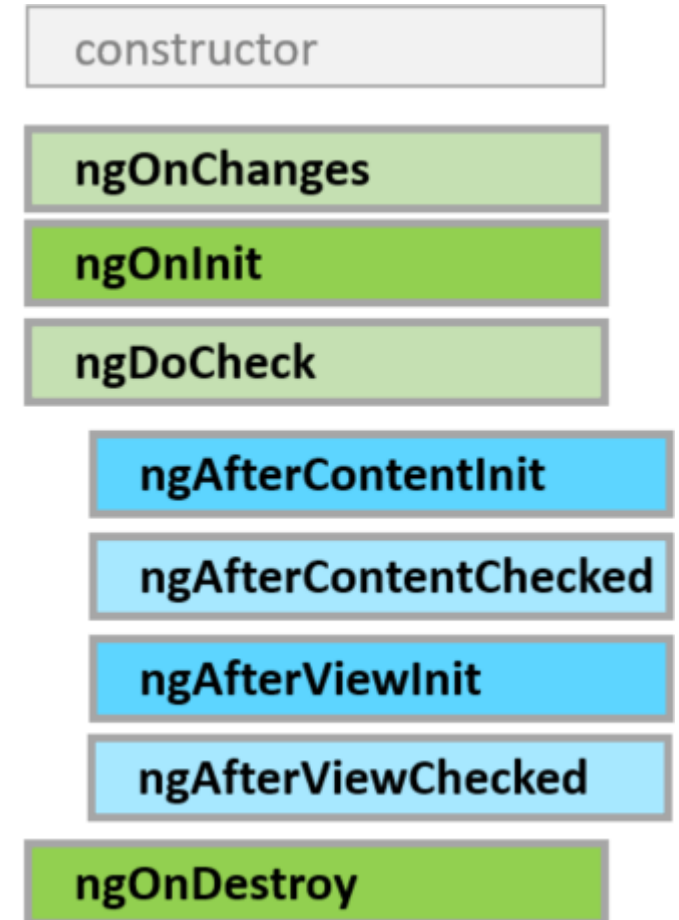
```
@Component({
  selector: 'comp',
  template: `
    <p>Message: {{message}}</p>
    <input [value]="message" (input)="message=$event.target.value">
  `
})
export class CompComponent {
  public message: string = 'Default Message';
}
```

Component Lifecycle Hooks

Angular components go through a multi-stage bootstrap and lifecycle process, and we can respond to various events as our app starts, runs, and creates/destroys components.

After Angular creates a component/directive by calling `new` on its constructor, it calls the lifecycle hook methods in the following sequence at specific moments.

Angular only calls a directive/component hook method if it is defined.



ngOnChanges Example

This component will be notified when its input properties change.

```
@Component({  
    selector: 'app-cmp'  
})  
class AppCmp implements OnChanges {  
    @Input() field1;  
    @Input() field2;  
  
    ngOnChanges(changes) {  
        //..  
    }  
}
```

Content and View children

A component can interact with its children. There are two types of children a component can have:


- Content child/children

- View child/children.

Template Local Variables

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-component',
  template: `<div>
    <input name="title" value="Asaad" #myName />
    <button (click)="sayMyName(myName)">Say my name</button>
  </div>`
})
export class MyComponent {
  @ViewChild('myName') myFullName;
  sayMyName(name) {
    console.log(`My name is ${name.value}`)
    // OR
    console.log(`My name is ${myFullName.nativeElement.value}`)
  }
}
```

myName is now an object that represents this input DOM element (HTMLInputElement).



Template Variables Binding

```
<comp> <p #myParentParagraph>Parent Paragraph</p> </comp>
```

```
import { Component, ContentChild } from '@angular/core';
@Component({
  selector: 'comp',
  template: `
    <ng-content></ng-content>
    <button (click)="getData()">Get Parent Paragraph Data</button> `
})
export class CompComponent {
  @ContentChild('myParentParagraph') myParentPObj;
  getData(){
    console.log(this.myParentPObj.nativeElement.textContent);
  }
}
```

Debugging Angular Apps

Angular DevTools extends Chrome DevTools adding Angular specific debugging and profiling capabilities.

You can use Angular DevTools to understand the structure of your application and preview the state of the directive and the component instances. To get insights into the execution of the application, you can use the profiler tab, which shows you the individual change detection cycles, what triggered them, and how much time Angular spent in them.

<https://blog.angular.io/introducing-angular-devtools-2d59ff4cf62f>

Main Points

Components in Angular are self-describing, they contain all the information needed to instantiate them. This means that any component can be bootstrapped. It does not have to be special in any way.

- A component knows how to interact with its host element.

- A component knows how to interact with its content and view children.

- A component knows how to render itself.

- A component configures dependency injection.

- A component has a well-defined public API of input and output properties.