**What are Components?**

In Angular, components are the building blocks of user interfaces. They are essentially TypeScript classes that encapsulate presentation logic, data binding, and lifecycle hooks. Each component defines a view, which is rendered as HTML, and can interact with other components.

**Manually Creating Components**

While Angular provides tools to generate components, understanding the manual process is crucial for deeper comprehension.

TypeScript

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

@Component({

selector: 'app-my-component',

template: `

<p>This is my component!</p>

`,

styles: [`

p {

color: blue;

}

`]

})

export class MyComponent {

// Component logic here

}

Use code [with caution.](https://d.docs.live.net/faq#coding)

* **@Component decorator:** This decorator defines the component metadata.
* **selector:** The HTML tag to use for this component.
* **template:** The component's template, written in Angular's template syntax.
* **styles:** The component's styles, written in CSS.

**Working with Component Templates and Component Styles**

**Component Templates:**

* Use Angular's template syntax to create dynamic views.
* Bind data to the view using interpolation, property binding, and event binding.
* Utilize directives to modify element behavior.
* Employ structural directives to control the DOM structure.

**Component Styles:**

* Encapsulate styles within the component using CSS.
* Utilize CSS preprocessors like Sass or Less for better organization.
* Apply CSS classes and styles dynamically using directives.

**Data Binding**

* **String Interpolation:** Embed data from the component class into the template using curly braces {{ }}.
* **Property Binding:** Bind component properties to element attributes or properties.
* **One-Way Binding:** Data flows from the component to the view.
* **Two-Way Binding:** Data flows in both directions (typically using [(ngModel)]).
* **Event Binding:** Bind event handlers to element events.

TypeScript

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

@Component({

selector: 'app-my-component',

template: `

<p>Name: {{ name }}</p>

<input type="text" [(ngModel)]="name">

<button (click)="onClick()">Click me</button>

`

})

export class MyComponent {

name = 'John Doe';

onClick() {

alert('Button clicked!');

}

}

Use code [with caution.](https://d.docs.live.net/faq#coding)

**Passing Data Between Components**

* **Input properties:** Pass data from parent to child components.
* **Output properties (Event Emitters):** Pass data from child to parent components.
* **Services:** Share data between components using a shared service.

**Advanced Topics**

* **Component Lifecycle Hooks:** ngOnInit, ngOnChanges, ngOnDestroy, etc.
* **Dependency Injection:** Inject services into components.
* **Reactive Forms:** Building complex forms with reactive programming.
* **Component Communication:** Using observables and subjects for complex data flow.

By mastering these concepts, you can create efficient and maintainable Angular applications.

**Data Binding in Angular: A Deep Dive**

Data binding is a core concept in Angular, facilitating the seamless synchronization of data between the component's TypeScript class (model) and the HTML template (view).

**String Interpolation**

* Embeds component property values directly into the template using double curly braces {{ }}.
* One-way binding from component to view.

TypeScript

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

@Component({

selector: 'app-interpolation',

template: `

<p>Hello, {{ name }}!</p>

`

})

export class InterpolationComponent {

name = 'John Doe';

}

Use code [with caution.](https://d.docs.live.net/faq#coding)

**Property Binding**

* Sets property values of DOM elements from component properties.
* One-way binding from component to view.
* Uses square brackets [] around the property name in the template.

TypeScript

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

@Component({

selector: 'app-property-binding',

template: `

<button [disabled]="isDisabled">Click me</button>

`

})

export class PropertyBindingComponent {

isDisabled = false;

}

Use code [with caution.](https://d.docs.live.net/faq#coding)

**One-Way Binding**

* Data flows from the component to the view.
* Includes string interpolation and property binding.
* Ideal for displaying data that doesn't change frequently.

**Two-Way Binding**

* Enables data flow in both directions: component to view and view to component.
* Typically used for form inputs.
* Achieved using [(ngModel)] directive.

TypeScript

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

@Component({

selector: 'app-two-way-binding',

template: `

<input type="text" [(ngModel)]="name">

<p>Your name is: {{ name }}</p>

`

})

export class TwoWayBindingComponent {

name = 'John Doe';

}

Use code [with caution.](https://d.docs.live.net/faq#coding)

**Event Binding**

* Handles user interactions with DOM elements.
* Uses parentheses () around the event name in the template.

TypeScript

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

@Component({

selector: 'app-event-binding',

template: `

<button (click)="onClick()">Click me</button>

`

})

export class EventBindingComponent {

onClick() {

console.log('Button clicked!');

}

}

Use code [with caution.](https://d.docs.live.net/faq#coding)

**Passing Data Between Components**

* **Input properties:** Pass data from parent to child components using @Input decorator.
* **Output properties (Event Emitters):** Pass data from child to parent components using @Output and EventEmitter.
* **Services:** Share data between components using a shared service.

TypeScript

// Child component

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

@Component({

selector: 'app-child',

template: `

<p>Child component: {{ data }}</p>

<button (click)="sendValue()">Send Value</button>

`

})

export class ChildComponent {

@Input() data: string;

@Output() valueChanged = new EventEmitter<string>();

sendValue() {

this.valueChanged.emit('Value from child');

}

}

Use code [with caution.](https://d.docs.live.net/faq#coding)

**Key Points:**

* Choose the appropriate binding type based on the data flow requirements.
* Use one-way binding for displaying data and property updates.
* Use two-way binding for form inputs and interactive elements.
* Consider event binding for user interactions.
* Utilize input and output properties for component communication.

**Data Binding in Angular: A Deep Dive**

Data binding is a core concept in Angular, facilitating the seamless synchronization of data between the component's TypeScript class (model) and the HTML template (view).

**String Interpolation**

* Embeds component property values directly into the template using double curly braces {{ }}.
* One-way binding from component to view.

TypeScript

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

@Component({

selector: 'app-interpolation',

template: `

<p>Hello, {{ name }}!</p>

`

})

export class InterpolationComponent {

name = 'John Doe';

}

Use code [with caution.](https://d.docs.live.net/faq#coding)

**Property Binding**

* Sets property values of DOM elements from component properties.
* One-way binding from component to view.
* Uses square brackets [] around the property name in the template.

TypeScript

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

@Component({

selector: 'app-property-binding',

template: `

<button [disabled]="isDisabled">Click me</button>

`

})

export class PropertyBindingComponent {

isDisabled = false;

}

Use code [with caution.](https://d.docs.live.net/faq#coding)

**One-Way Binding**

* Data flows from the component to the view.
* Includes string interpolation and property binding.
* Ideal for displaying data that doesn't change frequently.

**Two-Way Binding**

* Enables data flow in both directions: component to view and view to component.
* Typically used for form inputs.
* Achieved using [(ngModel)] directive.

TypeScript

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

@Component({

selector: 'app-two-way-binding',

template: `

<input type="text" [(ngModel)]="name">

<p>Your name is: {{ name }}</p>

`

})

export class TwoWayBindingComponent {

name = 'John Doe';

}

Use code [with caution.](https://d.docs.live.net/faq#coding)

**Event Binding**

* Handles user interactions with DOM elements.
* Uses parentheses () around the event name in the template.

TypeScript

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

@Component({

selector: 'app-event-binding',

template: `

<button (click)="onClick()">Click me</button>

`

})

export class EventBindingComponent {

onClick() {

console.log('Button clicked!');

}

}

Use code [with caution.](https://d.docs.live.net/faq#coding)

**Passing Data Between Components**

* **Input properties:** Pass data from parent to child components using @Input decorator.
* **Output properties (Event Emitters):** Pass data from child to parent components using @Output and EventEmitter.
* **Services:** Share data between components using a shared service.

TypeScript

// Child component

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

@Component({

selector: 'app-child',

template: `

<p>Child component: {{ data }}</p>

<button (click)="sendValue()">Send Value</button>

`

})

export class ChildComponent {

@Input() data: string;

@Output() valueChanged = new EventEmitter<string>();

sendValue() {

this.valueChanged.emit('Value from child');

}

}

Use code [with caution.](https://d.docs.live.net/faq#coding)

**Key Points:**

* Choose the appropriate binding type based on the data flow requirements.
* Use one-way binding for displaying data and property updates.
* Use two-way binding for form inputs and interactive elements.
* Consider event binding for user interactions.
* Utilize input and output properties for component communication.

By understanding these concepts and their practical applications, you can build efficient and reactive Angular applications.

**Would you like to explore advanced data binding scenarios or specific use cases?**

**Passing Data from Parent to Child Component in Angular**

**Using the @Input Decorator**

The most common method to pass data from a parent component to a child component is by using the @Input decorator.

**Parent Component:**

HTML

<app-child [data]="parentData"></app-child>

Use code [with caution.](https://d.docs.live.net/faq#coding)

**Child Component:**

TypeScript

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

@Component({

selector: 'app-child',

template: `

<p>Data from parent: {{ data }}</p>

`

})

export class ChildComponent {

@Input() data: any; // Replace 'any' with the appropriate type

}

Use code [with caution.](https://d.docs.live.net/faq#coding)

**Explanation:**

* The @Input() decorator in the child component defines a property data that can receive data from the parent.
* In the parent component's template, the [data] syntax binds the parentData property to the child component's data input property.

**Key Points**

* The @Input decorator creates a property in the child component that can be accessed and modified by the parent component.
* The data flow is one-way: from parent to child.
* The child component cannot modify the original data in the parent component.
* Use specific types for the @Input property to enhance type safety.

**Example with Type Safety**

**Understanding Directives in Angular**

Directives are classes that add new behavior to existing DOM elements. Angular provides two primary types:

**Structural Directives**

These directives modify the DOM structure by adding, removing, or replacing elements. They are prefixed with an asterisk (\*).

**ngIf**

Conditionally renders an element based on an expression.

HTML

<div \*ngIf="condition">

// Content to be shown if condition is true

</div>

Use code [with caution.](https://d.docs.live.net/faq#coding)

**ngIf with an else Condition**

You can use the else clause to render alternative content if the condition is false.

HTML

<div \*ngIf="condition; else elseBlock">

// Content to be shown if condition is true

</div>

<ng-template #elseBlock>

// Content to be shown if condition is false

</ng-template>

Use code [with caution.](https://d.docs.live.net/faq#coding)

**Attribute Directives**

These directives modify the behavior or appearance of an existing DOM element. They are prefixed with a square bracket [].

HTML

<button [disabled]="isDisabled">Click me</button>

Use code [with caution.](https://d.docs.live.net/faq#coding)

**Additional Directives**

Angular provides several built-in directives like ngFor, ngSwitch, ngClass, etc., each with its specific purpose.

**Key Points**

**Attributes and Structural Directives in Angular**

**Attributes Directives**

Attribute directives modify the behavior or appearance of an existing DOM element. They are prefixed with square brackets [] in the template.

**Common Attribute Directives:**

* **[disabled]**: Disables an element.
* **[style]**: Sets inline styles.
* **[class]**: Adds or removes CSS classes.
* **[ngClass]**: Dynamically adds or removes CSS classes based on conditions.
* **[ngStyle]**: Dynamically sets inline styles based on conditions.

**Example:**

HTML

<button [disabled]="isDisabled">Click me</button>

Use code [with caution.](https://d.docs.live.net/faq#coding)

**Structural Directives**

Structural directives modify the DOM structure by adding, removing, or replacing elements. They are prefixed with an asterisk \*.

**Common Structural Directives:**

* **\*ngIf**: Conditionally renders an element or template.
* **\*ngFor**: Creates a template instance for each item in an array.
* **\*ngSwitch**: Creates multiple templates and renders one based on a condition.

**Example:**

HTML

<div \*ngIf="condition">

// Content to be shown if condition is true

</div>

<ul>

<li \*ngFor="let item of items">{{ item }}</li>

</ul>

Use code [with caution.](https://d.docs.live.net/faq#coding)

**Key Differences**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Attribute Directives** | **Structural Directives** |
| Purpose | Modify element behavior or appearance | Modify DOM structure |
| Syntax | [] | \* |
| Examples | [disabled], [style], [ngClass] | \*ngIf, \*ngFor, \*ngSwitch |

Export to Sheets

**Additional Considerations**

* **Custom Directives:** You can create custom directives to encapsulate reusable behavior.
* **Directive Lifecycle Hooks:** Directives have lifecycle hooks like ngOnInit, ngOnChanges, and ngOnDestroy for performing actions at different stages.
* **Performance:** Be mindful of performance implications when using directives, especially with large datasets.