EXTRA FEATURES WITH COMPONENTS

Strong Typing & Interface

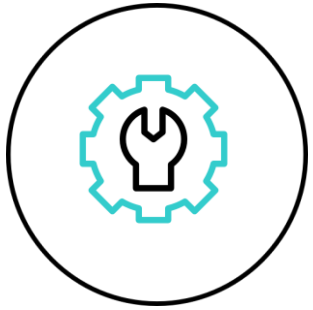
Strong typing is a programming language feature that enforces type safety. This means that variables must be declared with a specific type, and the compiler will check to make sure that they are only used with operations that are valid for that type. Interfaces are a way to define the structure of an object. They can be used to specify the properties and methods that an object must have.

Benefits of Strong Typing & Interfaces

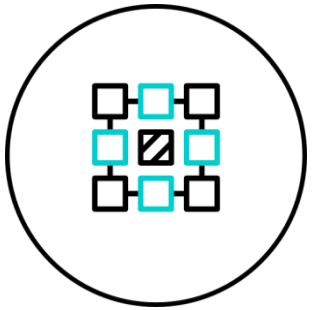
- ❑ Improves code quality and maintainability
- ❑ Reduces runtime errors
- ❑ Makes code easier to understand
- ❑ Improves developer productivity

Benefits of Strong Typing in Angular

Improved Code Reliability

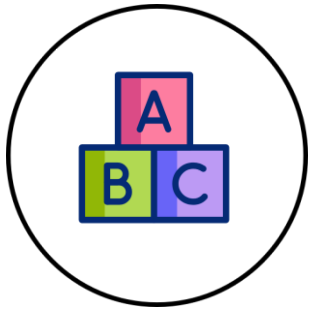


Enhanced Readability and Maintainability



IDE Support and Tooling

Catching Errors Early



Increased Confidence in Refactoring

Reduced Debugging Efforts

Interfaces

- Blueprints for data: Define the properties and their data types (e.g., name: string, age: number).
- Contracts for components: Specify the methods and functionalities a component must provide.
- Think of them as "rules of the game" for data consistency and clarity.

Example

```
interface Person {
name: string;
age: number;
email: string;
}
```


Styling in Angular: Methods and Approaches

- Inline Styles
- Component Styles
- External Stylesheets
- Global Styles

•Apply classes to elements in your HTML templates using the class attribute.

Inline Styles

- Add styles directly to HTML elements using the style attribute.
- Useful for quick styling or minor changes.
- Can become difficult to maintain for complex styles.

Component Styles

- Define styles within the component's TypeScript file using the styles array.
- More organized and maintainable than inline styles.
- Styles are scoped to the component, preventing conflicts.

External Stylesheets

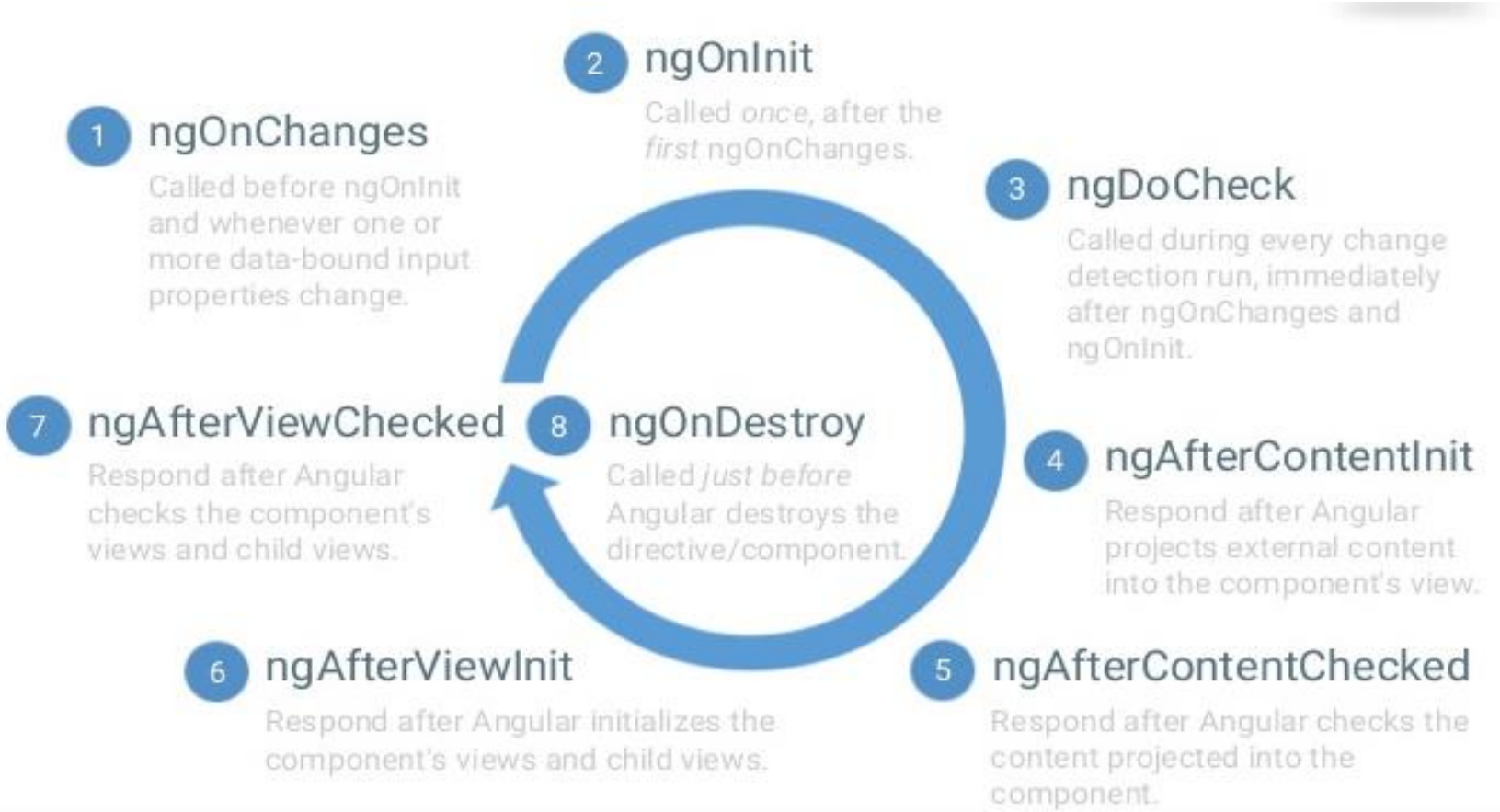
- Link to external CSS files from your index.html file.
- Useful for sharing styles across your entire application.
- Can lead to conflicts if not managed carefully.

Global Styles

- Define reusable CSS classes in a separate CSS file.
- Apply classes to elements in your HTML templates using the class attribute.
- Provides more flexibility and organization for complex styles.

Life Cycle Hook

Introducing the Lifecycle: Your Component's Journey in Angular



Boarding the Ride: Early Hooks (OnChanges, OnInit, DoCheck)

- **OnChanges:** This hook activates when a component's input properties change. Think of it as the car receiving instructions before the ride begins, adjusting based on new information.

- **OnInit:** Once the initial setup is complete, the car is ready to go! This hook gets called after the component and its children are initialized.

- **DoCheck:** This is like a vigilant conductor continuously checking the car's state. It runs during change detection cycles, giving you fine-grained control over data updates.

Mid-Ride Thrills: Content-Related Hooks (AfterContentInit, AfterViewChecked)

- **AfterContentInit:** The car has passed through the initial loop! This hook fires after the component's projected content is initialized, ideal for manipulating content from child components.
- **AfterViewChecked:** Time for the exciting twists and turns! This hook runs after changes are applied to the DOM, allowing you to react to rendered content or perform DOM manipulations.

Reaching the End: Final Checks and Unloading (OnDestroy)

OnDestroy:

The ride is over, and the car safely reaches the station. This hook runs just before the component is destroyed, allowing you to clean up resources, unsubscribe from subscriptions, and prevent memory leaks.

Custom Pipes in Angular

- Reusable functions that transform data in templates.
- Offer flexibility beyond built-in pipes.
- Encapsulate data formatting logic for maintainability.

Custom Pipes in Angular

1. Create a new TypeScript file (**e.g., my-custom-pipe.ts**).

2. Decorate the class with **@Pipe({ name: 'myCustomPipe' }).**

3.Implement the **PipeTransform** interface with a transform method.

4.Use the pipe in your template with the pipe name and arguments (e.g., **{{ value | myCustomPipe:arg1:arg2 }}).**

5. Inside transform, define the logic to transform the input data.

Custom Pipes

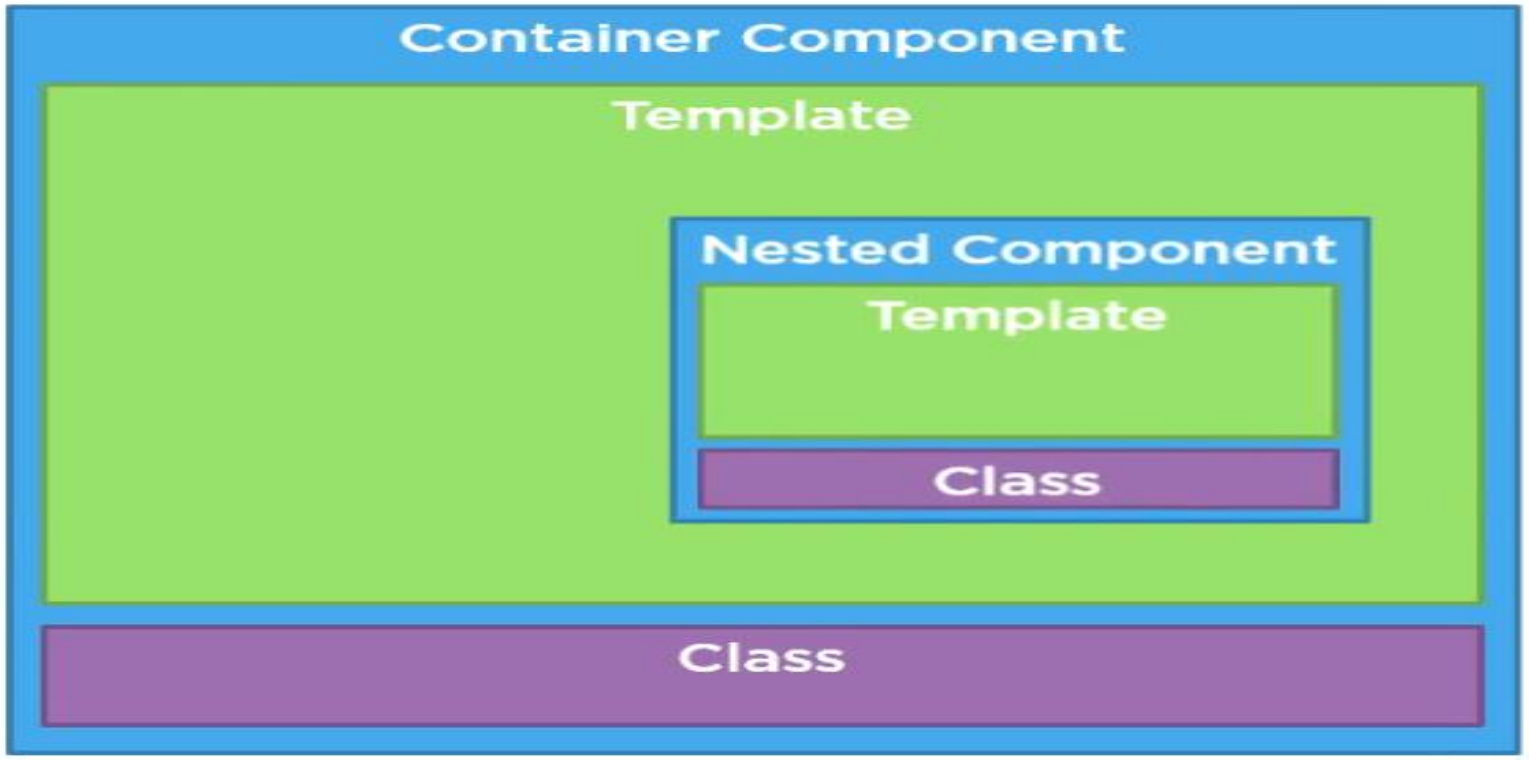
```
import { Pipe } from '@angular/core’;
@Pipe({
  name: 'ellipsis'
})
export class Ellipsis Pipe {
   transform(val, args)
   {
     if (args === undefined) {
       return val;
     }
     if (val.length > args) {
       return val.substring(0, args) + '...’;
     }
     else {
       return val;
     }
   }

 }
```


Nested Components

- Nesting a component means including a component inside another component.
- What Makes a component Nest-able? Its template only manages a fragment of a larger view.
- It has a selector.
- It optionally communicates with its container.

Nested Components

Add a nested component by using a component as a directive within another component.



Nested Components

Steps:

- Create Parent and Child Component
- Import the Child component into Parent Component
- Add the Child Component Class to Declaration part of Root module.
- Use the Child component tag inside Parent
- Component template

@Input & @Output decorators in Angular

@Input:

- Receives data from a parent component.
- Acts as a one-way binding, data flows only from parent to child.
- Think of it as a child listening to the parent's instructions.

@Output:

- Emits events from a child component to the parent.
- Enables two-way communication, the child can share information with the parent.
- Imagine the child raising its hand to get the parent's attention.

Example

Here's a complete and easily understandable example using @Input and @Output decorators for beginners:

Scenario: Imagine a simple product card component in an online store.

Parent Component (Product List):

- This component displays a list of product cards.
- Each card is a separate child component.
- Each card needs to display the product name and price. (This data comes from the parent component)
- When a user clicks the "Add to Cart" button on a card, the parent component needs to know (event needs to be sent).

Child Component (Product Card):

- This component receives the product name and price from the parent component using @Input.
- It displays the received name and price.
- When the user clicks the "Add to Cart" button, the component emits an event using @Output.

Parent Component

```
// product-list.component.ts

interface Product {
  name: string;
  price: number;
}

@Component({
  selector: 'app-product-list',
  template: `
    <app-product-card *ngFor="let product of products"
                      [name]="product.name"
                      [price]="product.price"
                      (addToCart)="onAddToCart($event)">
    </app-product-card>
  `,
})
export class ProductListComponent {
  products: Product[] = [
    { name: 'T-Shirt', price: 19.99 },
    { name: 'Mug', price: 9.99 },
  ];

  onAddToCart(product: Product) {
    // Handle adding product to cart logic here
    console.log('Product added to cart:', product);
  }
}
```

Child Component

```
// product-card.component.ts

@Component({
  selector: 'app-product-card',
  template: `
    <h2>{{ name }}</h2>
    <p>Price: ${{ price }}</p>
    <button (click)="addToCart()">Add to Cart</button>
  `,
})
export class ProductCardComponent {
  @Input() name!: string;
  @Input() price!: number;

  @Output() addToCart = new EventEmitter<any>();

  onClick() {
    this.addToCart.emit();
  }
}
```


Explanation

In the parent component, each product-card component has:

- [name] and [price] using @Input to receive data from the parent.
- (addToCart) using @Output to emit an event when the button is clicked.

In the child component:

- @Input() properties receive data from the parent.
- @Output() emits an event when the button is clicked.

Questions?

Aitrich