

# Introducción a las Directivas de Angular

# ¿Qué son las Directivas?

- Las directivas son un componente fundamental en Angular.
- Proporcionan una forma de extender y modificar el comportamiento de los elementos HTML existentes o crear nuevos elementos personalizados.
- Dos tipos principales de directivas en Angular:
  - Directivas estructurales: alteran la estructura del DOM.
  - Directivas de atributos: modifican el aspecto o el comportamiento de los elementos.

# Directivas Estructurales

- \*ngIf: muestra u oculta elementos basados en una condición.
- \*ngFor: repite elementos basados en una colección.
- \*ngSwitch: cambia el contenido basado en una expresión.

# Directiva \*ngIf

```
<div *ngIf="mostrarElemento">Este elemento se muestra si mostrarElemento es verdadero.</div>
```

- Muestra el elemento solo si la expresión mostrarElemento es verdadera.
- El elemento se elimina del DOM si la expresión es falsa.

# Directiva \*ngFor

```
<ul>  
  <li *ngFor="let item of listaItems">{{ item }}</li>  
</ul>
```

- Repite el elemento li para cada elemento en listaItems.
- La variable item contiene el valor del elemento actual en cada iteración.

# Directivas de Atributos

- `ngClass`: agrega o remueve clases CSS dinámicamente.
- `ngStyle`: agrega estilos CSS dinámicamente.
- `ngModel`: establece enlace bidireccional entre una propiedad y un control de formulario.

# Directiva ngClass

```
<div [ngClass]="{ 'clase1': condicion1, 'clase2': condicion2 }">...</div>
```

- Agrega la clase CSS clase1 si condicion1 es verdadera, y clase2 si condicion2 es verdadera.
- Puedes combinar múltiples condiciones y clases.

# Directiva ngStyle

```
<div [ngStyle]="{ 'color': colorTexto, 'font-size': tamanoFuente }">...</div>
```

- Aplica estilos CSS en línea al elemento.
- Los estilos se aplican si las propiedades colorTexto y tamanoFuente son válidas.



# Creando Directivas Personalizadas

- Angular permite crear directivas personalizadas.
- Una directiva personalizada extiende el comportamiento de los elementos HTML existentes o crea nuevos elementos personalizados.

# Creando Directivas Personalizadas (ejemplo)

```
@Directive({  
  selector: '[miDirectiva]'  
})  
export class MiDirectiva {  
  // Lógica de la directiva...  
}
```

```
<div miDirectiva>Este elemento usa una directiva personalizada.</div>
```

- La directiva personalizada se usa agregando el atributo miDirectiva al elemento HTML.

# Directiva Personalizada - *appHighlight*

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class AppHighlightDirective {
  @Input('appHighlight') condition: boolean;

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef
  ) { }

  ngOnInit() {
    if (this.condition) {
      this.viewContainer.createEmbeddedView(this.templateRef);
    } else {
      this.viewContainer.clear();
    }
  }
}
```



theme: base-colors  
class: invert

## Directiva Personalizada - appColor

- Vamos a crear una directiva de atributo personalizada llamada `appColor`.
- Esta directiva se utilizará para aplicar un color de fondo personalizado a un elemento HTML.

# Paso 1: Crear la directiva

```
import { Directive, ElementRef, Input, OnInit } from '@angular/core';

@Directive({
  selector: '[appColor]'
})
export class AppColorDirective implements OnInit {
  @Input('appColor') color: string;

  constructor(private elementRef: ElementRef) {}

  ngOnInit() {
    this.elementRef.nativeElement.style.backgroundColor = this.color;
  }
}
```

## Paso 2: Usar la directiva

```
<div appColor="red">Este elemento tiene un color de fondo rojo.</div>
```



# Accediendo a Componentes desde el Código en Angular

# ¿Por qué necesitamos acceder a los componentes?

- En ocasiones, es necesario acceder a los componentes desde el código para realizar tareas como:
  - Obtener o modificar valores de propiedades.
  - Ejecutar métodos o funciones.
  - Comunicarse entre componentes.
- Angular proporciona varias formas de acceder a los componentes desde el código.

# Accediendo a componentes mediante ViewChild

- Utilizamos la directiva `@ViewChild` para acceder a un componente desde el código.

```
import { Component, ViewChild } from '@angular/core';
import { MyComponent } from './my-component';

@Component({
  selector: 'app-parent',
  template: `
    <app-my-component></app-my-component>
  `
})
export class ParentComponent {
  @ViewChild(MyComponent) myComponent: MyComponent;

  ngAfterViewInit() {
    // Acceso al componente hijo
    console.log(this.myComponent);
  }
}
```

# Accediendo a componentes mediante ElementRef

- Utilizamos ElementRef para acceder al elemento HTML de un componente desde el código.

# Pipes en Angular

# ¿Qué son las Pipes?

- Las Pipes son una característica de Angular que nos permite transformar los datos antes de mostrarlos en la plantilla.
- Proporcionan una sintaxis concisa para aplicar transformaciones comunes a los datos.

# Tipos de Pipes incorporadas

- Angular proporciona una variedad de Pipes incorporadas, como:
  - **UpperCasePipe**: Convierte el texto en mayúsculas.
  - **LowerCasePipe**: Convierte el texto en minúsculas.
  - **CurrencyPipe**: Formatea un número como una moneda.
  - **DatePipe**: Formatea una fecha.
  - **DecimalPipe**: Formatea un número con decimales.

# Uso de las Pipes

- Las Pipes se pueden utilizar en las plantillas de los componentes de Angular de la siguiente manera:

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

@Component({
  selector: 'app-pipes-example',
  template: `
    <h2>PIPES EN ANGULAR</h2>

    <h4>Texto: {{ texto }}</h4>
    <p>Mayúsculas: {{ texto | uppercase }}</p>
    <p>Minúsculas: {{ texto | lowercase }}</p>

    <h4>Número: {{ numero }}</h4>
    <p>Decimal: {{ numero | number }}</p>
    <p>Moneda: {{ numero | currency }}</p>
    <p>Porcentaje: {{ numero | percent }}</p>

    <h4>Fecha: {{ fecha }}</h4>
    <p>Fecha corta: {{ fecha | date:'short' }}</p>
    <p>Fecha larga: {{ fecha | date:'long' }}</p>

    <h4>Arreglo: {{ arreglo }}</h4>
    <p>Ordenado: {{ arreglo | sort }}</p>
    <p>Reverso: {{ arreglo | reverse }}</p>

    <h4>Objeto: {{ objeto | json }}</h4>
  `
})
export class PipesExampleComponent {
  texto: string = 'Hola, Mundo!';
```



# ¿Qué son las Pipes Personalizadas?

- Las pipes personalizadas son funciones que nos permiten crear transformaciones personalizadas de datos en Angular.

# Creando una Pipe Personalizada

- Para crear una pipe personalizada, debemos seguir estos pasos:
  1. Crear una clase y decorarla con `@Pipe`.
  2. Implementar la interfaz `PipeTransform`.
  3. Definir el método `transform()` para realizar la transformación deseada.

# Ejemplo de Pipe Personalizada

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'capitalized'
})
export class CapitalizedPipe implements PipeTransform {
  transform(value: string): string {
    if (!value) return '';

    const words = value.split(' ');
    const capitalizedWords = words.map(word => {
      return word.charAt(0).toUpperCase() + word.slice(1);
    });

    return capitalizedWords.join(' ');
  }
}
```

# **RxJS: Programación Reactiva en Angular**

# ¿Qué es RxJS?

- RxJS (Reactive Extensions for JavaScript) es una biblioteca para programación reactiva en JavaScript.
- Proporciona una forma elegante y poderosa de trabajar con flujos de datos asíncronos.

# Conceptos Básicos de RxJS

- **Observables:** Representan una fuente de datos que puede emitir valores a lo largo del tiempo.
- **Operadores:** Permiten transformar, filtrar y combinar observables.
- **Suscripción:** Permite escuchar los valores emitidos por un observable.
- **Subject:** Permite actuar como un observable y emitir valores manualmente.

# Ejemplo de Uso de RxJS

```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-rxjs-example',
  template: `
    <h2>RxJS en Acción</h2>
    <p>{{ message }}</p>
  `
})
export class RxJSExampleComponent implements OnInit {
  message: string = '';

  ngOnInit() {
    const source$ = new Observable(observer => {
      setTimeout(() => {
        observer.next('Hola, RxJS!');
        observer.complete();
      }, 2000);
    });

    source$.subscribe(value => {
      this.message = value;
    });
  }
}
```

# ¿Qué son los Pipes en RxJS?

- Los pipes en RxJS son operadores que nos permiten transformar y manipular los flujos de datos.



# Ejemplo de Uso de Pipes en RxJS

- En este ejemplo, utilizamos el operador `map` para transformar los valores de un flujo de datos:

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

const numbers$ = of(1, 2, 3, 4, 5);

numbers$.pipe(
  map(number => number * 2)
).subscribe(result => {
  console.log(result); // 2, 4, 6, 8, 10
});
```

# Algunos Operadores de Pipes en RxJS

- RxJS proporciona una amplia variedad de operadores de pipes para transformar y manipular los flujos de datos.

# map

- **Descripción:** Transforma cada valor emitido aplicando una función.
- **Ejemplo:**

```
source$.pipe(  
  map(value => value * 2)  
)  
.subscribe(result => {  
  console.log(result); // Valores multiplicados por 2  
});
```

