Unidad 2: Fundamentos de Angular

Fundamentos de TypeScript

Introducción a TypeScript

Es un lenguaje de programación de código abierto desarrollado por Microsoft. Se describe como un "superset" de JavaScript, lo que significa que cualquier código JavaScript válido también es código TypeScript. Este lenguaje agrega características adicionales como el tipado estático, interfaces, y la programación orientada a objetos (POO), haciéndolo especialmente útil para proyectos grandes y complejos.

Ventajas de TypeScript en proyectos Angular

1. **Detección temprana de errores**:
   - El tipado estático permite encontrar errores en tiempo de desarrollo, evitando posibles fallos en producción.
   - Por ejemplo, asignar un número a una variable que debería ser un string generará un error antes de ejecutar el programa.
2. **Autocompletado y herramientas mejoradas**:
   - Los editores modernos, como Visual Studio Code, aprovechan las características de TypeScript para ofrecer autocompletado, refactorización segura y navegación fácil por el código.
3. **Escalabilidad**:
   - TypeScript es ideal para proyectos grandes donde se requiere un código mantenible y estructurado.
4. **Compatibilidad con JavaScript**:
   - TypeScript se transcompila a JavaScript, asegurando que pueda ejecutarse en cualquier entorno compatible con JavaScript.

Tipos en TypeScript

Los tipos son fundamentales en TypeScript y ayudan a garantizar que el código sea consistente y claro.

- `string`: Cadenas de texto.

  ```
  let saludo: string = "Hola, TypeScript";
  ```
- `number`: Números enteros o decimales.

```
let numero: number = 42;
```

- **boolean**: Valores verdadero o falso.

```
let activo: boolean = true;
```

- **any**: Permite cualquier tipo de dato, útil en situaciones donde el tipo no se conoce de antemano.

```
let desconocido: any = "Texto o número";
```

- **void**: Usado en funciones que no retornan un valor.

```
function saludar(): void {
console.log("Hola");
}
```

- **unknown**: Similar a `any`, pero requiere una verificación de tipo antes de usar el valor.

```
let dato: unknown = "Texto";

if (typeof dato === "string") {
    console.log(dato.toUpperCase());
}
```

Interfaces en TypeScript

Las interfaces definen la estructura que deben seguir los objetos. Son útiles para validar que un objeto tenga propiedades específicas.

```
interface Persona {
    nombre: string;
    edad: number;
}

let usuario: Persona = { nombre: "Juan", edad: 25 };
```

Clases en TypeScript

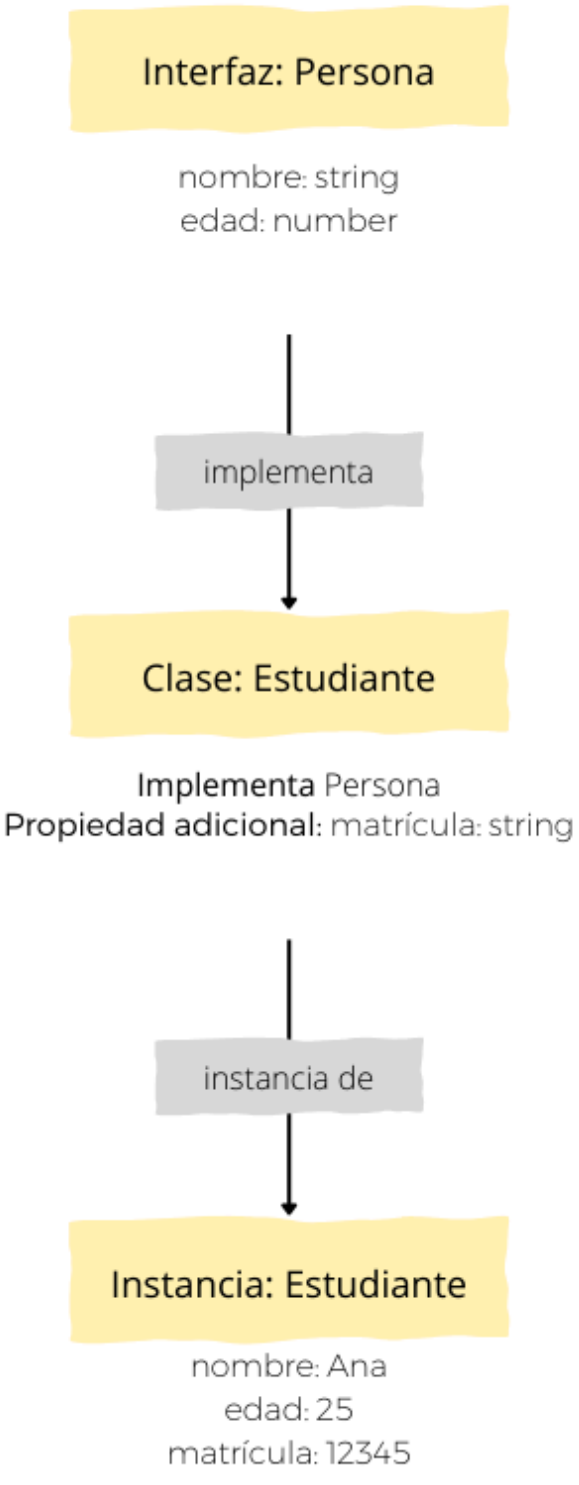
Las clases permiten usar programación orientada a objetos. Soportan conceptos como herencia, encapsulación y polimorfismo.

```
class Vehiculo {
    constructor(public marca: string, public modelo: string) {}
}

class Coche extends Vehiculo {
    constructor(marca: string, modelo: string, public puertas: number)
{
        super(marca, modelo);
```

```
    }
}

let miCoche = new Coche("Toyota", "Corolla", 4);
```

Transcompilación a JavaScript

TypeScript necesita ser transcompilado a JavaScript para ejecutarse en navegadores o entornos como Node.js. Esto se realiza mediante el compilador de TypeScript (`tsc`).



Tipos y estructuras de datos

TypeScript amplía los tipos básicos de JavaScript con características avanzadas, proporcionando mayor control sobre los datos.

Tipos avanzados

1. **Tuplas**: Permiten definir un arreglo con un número fijo de elementos, cada uno con un tipo específico.

   ```
   let coordenadas: [number, number] = [10, 20];
   ```

2. **Enums**: Representan un conjunto de valores constantes. Los valores pueden ser numéricos o de texto.

   ```
   enum Color {
  Rojo = "Rojo",
  Verde = "Verde",
  Azul = "Azul"
   }
   let colorFavorito: Color = Color.Rojo;
   ```

3. **Union Types**: Permiten que una variable acepte múltiples tipos.

   ```
   let id: string | number;
   id = 123;
   id = "ABC123";
   ```

4. **Tipo literal**: Define valores específicos para una variable.

   ```
   let direccion: "Norte" | "Sur" | "Este" | "Oeste";
   direccion = "Norte";
   ```

Tipos inferidos

TypeScript puede inferir automáticamente el tipo de una variable en función del valor asignado.

```
let nombre = "Carlos"; // TypeScript infiere que es de tipo string.
```

Introducción a Node.js y npm

¿Qué es Node.js?

Node.js es una tecnología de servidor que permite ejecutar JavaScript en el lado del servidor. A pesar de ser una tecnología principalmente de back-end, su importancia en el desarrollo de aplicaciones Angular radica en la necesidad de contar con herramientas clave que se instalan a través de Node.js.

Puntos clave:

- **Ejecución de JavaScript en el servidor:** Permite la ejecución de código JavaScript fuera del navegador.

- **Entorno de desarrollo:** Node.js se utiliza para instalar y gestionar paquetes esenciales para trabajar con Angular.

¿Por qué Node.js es esencial para Angular?

Si bien Angular no se ejecuta en Node.js, su ecosistema de desarrollo depende en gran medida de este entorno para la instalación de dependencias y la ejecución de comandos del Angular CLI.

Razones clave:

- **Instalación de Angular CLI:** La herramienta Angular CLI se instala con `npm`, que proviene de Node.js.
- **Gestión de dependencias:** A través de npm, se instalan bibliotecas y paquetes necesarios para el desarrollo de una aplicación Angular.
- **Entorno de desarrollo:** Angular CLI usa comandos de Node.js y npm para compilar, ejecutar pruebas y preparar la aplicación para producción.

Nota: Aunque Node.js se usa durante el desarrollo, las aplicaciones Angular no requieren Node.js para ejecutarse en producción.

Flujo de trabajo con Node.js en un proyecto Angular

El flujo de trabajo con Node.js en un proyecto Angular incluye varios pasos clave:

1. Instalación de Node.js

- Se instala Node.js desde nodejs.org.
- Con la instalación de Node.js, se obtiene el acceso al administrador de paquetes npm (Node Package Manager).

2. Creación de un proyecto Angular

- Con Angular CLI instalado (gracias a npm), se crea un proyecto Angular con el comando:

```
ng new nombre-del-proyecto
```

3. Gestión de dependencias

- La creación del proyecto genera un archivo `package.json`, que contiene todas las dependencias necesarias para el proyecto.
- Las dependencias se instalan ejecutando:

```
npm install
```

4. Ejecución del servidor de desarrollo

- Angular se ejecuta en un servidor de desarrollo local mediante:

```
ng serve
```

5. Compilación para producción

- La aplicación se compila para producción con:

```
ng build --prod
```

Archivos esenciales de Node.js en un proyecto Angular

1. package.json

Este archivo actúa como "manifiesto" del proyecto. Contiene información clave, como:

- **Nombre y versión del proyecto**
- **Dependencias**: Listado de bibliotecas necesarias para que la aplicación funcione.
- **Scripts de comandos**: Comandos predefinidos para ejecutar tareas comunes, como pruebas, compilaciones y ejecución.

Comandos clave de npm:

```
npm init         # Inicializa un nuevo package.json
npm init --yes   # Crea un package.json con valores predeterminados
```

Ejemplo de package.json:

```
{
  "name": "mi-proyecto-angular",
  "version": "1.0.0",
  "scripts": {
    "start": "ng serve",
    "build": "ng build"
  },
  "dependencies": {
    "@angular/core": "^15.0.0",
    "rxjs": "^7.0.0"
  },
  "devDependencies": {
    "@angular/cli": "^15.0.0",
    "typescript": "^4.0.0"
  }
}
```

Comandos esenciales de Node.js para Angular

Comando	Descripción
npm init	Crea un archivo package.json inicial.
npm install	Instala las dependencias de un proyecto.
npm start	Ejecuta el script de inicio definido en package.json.

npm run build	Ejecuta el script de compilación definido en package.json.
ng new	Crea un nuevo proyecto Angular.
ng serve	Sirve la aplicación en el servidor de desarrollo.
ng build	Compila la aplicación para producción.

Decoradores en TypeScript

Los **decoradores** son funciones especiales que se utilizan para **modificar clases, métodos, propiedades o parámetros**. Angular utiliza decoradores para definir componentes, directivas y servicios.

Tipos de Decoradores

1. **Decorador de Clase**

```
function Log(clase: Function) {
  console.log(`La clase ${clase.name} ha sido declarada.`);
}

@Log
class MiClase {}
```

Explicación: El decorador `@Log` imprime un mensaje cuando la clase `MiClase` es creada.

2. **Decorador de Propiedad**

```
function Capitalizar(target: any, key: string) {
  let valor = target[key];

  const getter = () => valor.toUpperCase();
  const setter = (nuevoValor: string) => valor = nuevoValor;

  Object.defineProperty(target, key, { get: getter, set: setter
});
}

class Usuario {
  @Capitalizar
  nombre: string = "juan";
}

const usuario = new Usuario();
console.log(usuario.nombre); // JUAN
```

Explicación: El decorador **@Capitalizar** transforma la propiedad **nombre** a mayúsculas.

3. **Decorador de Método**

```
function Autobind(_: any, _2: string, descriptor:
PropertyDescriptor) {
  const metodoOriginal = descriptor.value;
  const nuevoDescriptor: PropertyDescriptor = {
    configurable: true,
    enumerable: false,
    get() {
      const boundFn = metodoOriginal.bind(this);
      return boundFn;
    }
  };
  return nuevoDescriptor;
}

class Boton {
  mensaje = "Haz clic en mí";

  @Autobind
  mostrarMensaje() {
    console.log(this.mensaje);
  }
}

const boton = new Boton();
const botonHTML = document.querySelector('button');
botonHTML?.addEventListener('click', boton.mostrarMensaje);
```

Explicación: El decorador **@Autobind** asegura que `this` en el método **mostrarMensaje** siempre se refiera a la clase **Boton**.

Módulos y componentes

Sistema de módulos

El sistema de módulos organiza el código en bloques reutilizables. Cada archivo en TypeScript puede actuar como un módulo.

Exportación e importación

- **Exportar elementos**:

```
  export function saludar(nombre: string): string {
return `Hola, ${nombre}`;
```

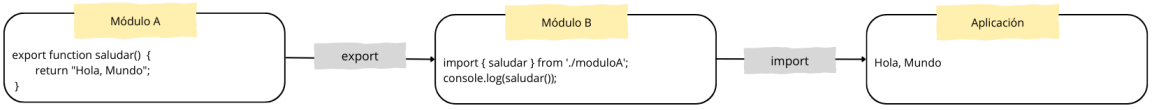
```
}
```

- **Importar elementos**:

```
import { saludar } from './utilidades';

console.log(saludar("Mundo"));
```

Ventajas del sistema de módulos

1. **Organización**: Facilita la separación de responsabilidades.
2. **Reutilización**: Permite usar el mismo módulo en múltiples partes del proyecto.
3. **Mantenimiento**: El código modular es más fácil de depurar y actualizar.



Componentización en Angular

En Angular, las aplicaciones están organizadas en **componentes**, que encapsulan lógica, vistas y estilos.

Partes de un componente

1. **Archivo `.ts`**: Contiene la lógica y los datos del componente.

```
  export class AppComponent {
titulo: string = "Mi Aplicación Angular";
  }
```

2. **Archivo `.html`**: Define la vista del componente.

```
<h1>{{ titulo }}</h1>
```

3. **Archivo `.css`**: Contiene los estilos específicos del componente.

Conexión modelo - vista

Interpolación en Angular

La interpolación es una técnica en Angular que conecta el modelo de datos de un componente con la vista HTML. Esto permite mostrar información dinámica directamente en el DOM.

La interpolación utiliza doble llave (`{{ }}`) para evaluar expresiones en el contexto del componente y mostrar el resultado en la vista. Puede usarse para:

- Mostrar valores de variables.
- Realizar operaciones simples.
- Invocar métodos del componente que retornan valores.

Características clave

1. **Unidireccionalidad**: La interpolación es un flujo unidireccional: desde el componente hacia la vista. Esto significa que los cambios en el modelo se reflejan automáticamente en la vista, pero no al revés.
2. **Seguridad**: Angular sanitiza automáticamente las expresiones interpoladas para evitar vulnerabilidades como ataques de inyección de scripts (XSS).
3. **Flexibilidad**: Puedes usar expresiones complejas dentro de las llaves para calcular resultados en tiempo de ejecución.

Ejemplos prácticos

Mostrar valores simples

```
// componente.ts
export class InterpolacionComponent {
    titulo: string = "Bienvenido a Angular";
}
```

_

```
<!-- componente.html -->
<h1>{{ titulo }}</h1>
```

Resultado en el DOM:

```
<h1>Bienvenido a Angular</h1>
```

Operaciones aritméticas

```
export class InterpolacionComponent {
    precio: number = 100;
    descuento: number = 20;
}
```

_

```
<p>Precio final: {{ precio - descuento }}</p>
```

Resultado en el DOM:

```
<p>Precio final: 80</p>
```

Llamar a métodos

```
export class InterpolacionComponent {
    obtenerFechaActual(): string {
        return new Date().toLocaleDateString();
    }
}
```

_

```
<p>Fecha de hoy: {{ obtenerFechaActual() }}</p>
```

Property Binding

El Property Binding es una técnica en Angular que permite enlazar propiedades del componente con atributos del DOM. Se utiliza para establecer valores dinámicos en propiedades HTML y atributos específicos.

¿Qué es el Property Binding?

Es una forma de comunicación unidireccional entre el modelo de datos del componente y la vista. Utiliza corchetes (`[ ]`) para enlazar propiedades del DOM con expresiones en el componente.

Diferencias entre interpolación y Property Binding

- **Interpolación**: Se usa principalmente para insertar texto o contenido en elementos HTML.
- **Property Binding**: Se utiliza para establecer propiedades del DOM que no se manejan como texto, como valores de atributos, estados de elementos o estilos.

Sintaxis

```
<!-- Property Binding -->
<elemento [propiedad]="expresion"></elemento>
```

Ejemplos prácticos

Enlace de atributos
typescript

```
export class PropertyBindingComponent {
    imagenUrl: string = "logo.png";
    altTexto: string = "Logo de la empresa";
}
```

html

```
<img [src]="imagenUrl" [alt]="altTexto">
```

Resultado en el DOM:

html

```
<img src="logo.png" alt="Logo de la empresa">
```

Habilitar o deshabilitar elementos
typescript

```
export class PropertyBindingComponent {
    habilitado: boolean = false;
}
```

html

```
<button [disabled]="!habilitado">Enviar</button>
```

Resultado en el DOM (cuando `habilitado` es `false`):

html

```
<button disabled>Enviar</button>
```

Directivas Angular

Directivas estructurales

Las directivas estructurales alteran el diseño del DOM al agregar, eliminar o modificar elementos dinámicamente. Angular ofrece directivas predefinidas como `*ngIf` y `*ngFor` para trabajar con condiciones y listas.

¿Qué son las directivas estructurales?

Son una clase de directivas que modifican la estructura del DOM basándose en el modelo de datos. Se identifican por el prefijo `*`.

Tipos de directivas estructurales

1. **`*ngIf`**: Permite mostrar o esconder elementos condicionalmente.

```
  export class DirectivasComponent {
mostrar: boolean = true;
  }
```

```
  <div *ngIf="mostrar">Este contenido se muestra si 'mostrar' es
  verdadero</div>
```

2. **`*ngFor`**: Itera sobre una lista para crear múltiples elementos.

```
  export class DirectivasComponent {

nombres: string[] = ["Ana", "Luis", "Carlos"];
  }
```

```
  <ul>
      <li *ngFor="let nombre of nombres">{{ nombre }}</li>
  </ul>
```

3. **`*ngSwitch`**: Se usa para mostrar un elemento basado en una condición específica.

```
  export class DirectivasComponent {
```

```
estado: string = "activo";
  }
```

```
  <div [ngSwitch]="estado">
      <p *ngSwitchCase="'activo'">El usuario está activo</p>
      <p *ngSwitchCase="'inactivo'">El usuario está inactivo</p>
      <p *ngSwitchDefault>Estado desconocido</p>
  </div>
```

Directivas de atributo

Las directivas de atributo alteran el aspecto o comportamiento de elementos HTML existentes sin cambiar la estructura del DOM.

¿Qué son las directivas de atributo?

Son directivas que se aplican a un elemento, componente o directiva existente para modificar sus propiedades, estilos o clases dinámicamente.

Directivas de atributo más comunes

1. **`[ngClass]`**: Permite asignar clases CSS dinámicamente en función de condiciones.

   typescript
   ```
   export class DirectivasAtributoComponent {
  hayError: boolean = true;
  }
   ```
   html
   ```
   <div [ngClass]="{ 'error': hayError, 'exito': !hayError
   }">Estado</div>
   ```

2. **`[ngStyle]`**: Permite asignar estilos CSS dinámicamente.

   typescript
   ```
   export class DirectivasAtributoComponent {
  tamaño: boolean = true;
   }
   ```
   html
   ```
   <div [ngStyle]="{ 'font-size': tamaño ? '20px' : '12px' }">
       Texto dinámico
   </div>
   ```

Filtros de vista

Introducción a Pipes

¿Qué es un Pipe?

Los **pipes** (filtros) son herramientas que permiten transformar la forma en que se presentan los datos en la vista de una aplicación Angular. En lugar de modificar el modelo de datos directamente, los pipes cambian **solo la forma en que los datos se ven en la interfaz de usuario (UI)**.

Los pipes son **fáciles de usar** y se aplican en las vistas HTML con la siguiente sintaxis:

{{ valor | pipe }}

Donde:

- **valor**: Es el dato que será transformado.
- **pipe**: Es el nombre del pipe que se aplicará a la transformación.

¿Para qué se utilizan los Pipes en Angular?

Los pipes se utilizan principalmente para:

- **Formatear texto**: Convertir texto en mayúsculas, minúsculas, etc.
- **Dar formato a números**: Mostrar números con decimales, como porcentajes o valores monetarios.
- **Formatear fechas**: Convertir fechas en formatos más legibles (dd/MM/yyyy, etc.).
- **Aplicar lógica de transformación**: Transformar la forma en que se presenta la información (ejemplo, mostrar solo los primeros caracteres de una cadena).

Sintaxis básica de los Pipes.

La sintaxis para usar un pipe es simple y clara:

```
{{ valor | nombreDelPipe }}
```

- **valor**: Dato que se desea transformar.
- **nombreDelPipe**: Nombre del pipe que se aplica a la transformación.

Ejemplo práctico:

```
<p>{{ 'angular' | uppercase }}</p> <!-- Resultado: ANGULAR -->
```

Pipes más utilizados

1. Pipe uppercase

Este pipe convierte una cadena de texto a **mayúsculas**.

Sintaxis

{{ texto | uppercase }}

Ejemplo práctico

{{ 'angular' | uppercase }} <!-- Resultado: ANGULAR -->

¿Cuándo usarlo?

- Para mostrar títulos, encabezados o nombres de forma destacada.
- Para seguir convenciones de estilo (por ejemplo, nombres de productos en mayúsculas).

2. Pipe lowercase

Este pipe convierte una cadena de texto a **minúsculas**.

Sintaxis

```
{{ texto | lowercase }}
```

Ejemplo práctico

```
<p>{{ 'ANGULAR' | lowercase }}</p> <!-- Resultado: angular -->
```

¿Cuándo usarlo?

- Para normalizar la entrada de datos antes de realizar comparaciones.
- Para presentar texto de forma sencilla y sin capitalización.

3. Pipe data

Este pipe transforma objetos de tipo **Date** en una fecha legible. Se pueden personalizar los formatos de la fecha.

Sintaxis

```
{{ fecha | date: 'formato' }}
```

Donde:

- **fecha**: El valor de tipo `Date` a formatear.
- **formato**: El formato en que se desea mostrar la fecha (por ejemplo, **dd/MM/yyyy**, **MM/dd/yyyy**, etc.).

Ejemplo práctico

```
<p>{{ fecha | date:'dd/MM/yyyy' }}</p>
<!-- Resultado: 05/04/2024 (suponiendo que la fecha sea 5 de abril de 2024) -->
```

¿Cuándo usarlo?

- Para mostrar fechas de creación, actualización o vencimiento.
- En informes, tickets o facturas que requieran una fecha de referencia.

4. Pipe currency

Este pipe transforma números en formato de **moneda**.

Sintaxis

```
{{ valor | currency: 'códigoMoneda': 'símbolo' }}
```

Donde:

- **valor**: El número a formatear.
- **códigoMoneda**: El código ISO de la moneda, como **USD** (dólares) o **EUR** (euros).
- **símbolo**: Indica si se muestra el símbolo (por ejemplo, **$**) o el código (**USD**).

Ejemplo práctico

```
<p>{{ 1500 | currency:'EUR' }}</p> <!-- Resultado: €1.500,00 -->
<p>{{ 1500 | currency:'USD' }}</p> <!-- Resultado: $1,500.00 -->
```

¿Cuándo usarlo?

- Para mostrar precios de productos o servicios.
- Para informes financieros o facturas.

5. Pipe percent

Este pipe transforma números decimales en porcentajes.

Sintaxis

```
{{ valor | percent: 'formato' }}
```

Donde:

- **valor**: El número decimal que se convertirá en un porcentaje.
- **formato**: Opcional, indica la cantidad de decimales a mostrar.

Ejemplo práctico

```
<p>{{ 0.75 | percent }}</p> <!-- Resultado: 75% -->
<p>{{ 0.75 | percent:'1.0-2' }}</p> <!-- Resultado: 75.00% -->
```

¿Cuándo usarlo?

- Para mostrar porcentajes en gráficos de progreso o informes.
- Para representar descuentos o tasas

Actividades de la clase práctica en vivo - Unidad 2

Actividad 1: Creación de un componente básico

Duración: 15 minutos

Objetivo: Aprender a crear y configurar un componente utilizando Angular CLI, definiendo propiedades y conectándolas a la vista.

Instrucciones:

1. Usar Angular CLI para generar un nuevo componente llamado **perfil-usuario**.
2. En el archivo **.ts**, agregar las siguientes propiedades:
    - **nombre**: String
    - **edad**: Number
    - **ocupacion**: String
3. En el archivo **.html**, mostrar los valores de estas propiedades utilizando **interpolación**.

Resolución:

```
// perfil-usuario.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-perfil-usuario',
  templateUrl: './perfil-usuario.component.html',
  styleUrls: ['./perfil-usuario.component.css']
})
export class PerfilUsuarioComponent {
  nombre: string = 'Juan Pérez';
  edad: number = 30;
  ocupacion: string = 'Desarrollador web';
}
```

```
<!-- perfil-usuario.component.html -->
<h1>Perfil de Usuario</h1>
<p>Nombre: {{ nombre }}</p>
<p>Edad: {{ edad }}</p>
<p>Ocupación: {{ ocupacion }}</p>
```

Actividad 2. Aplicando TypeScript

Duración: 10 minutos

Objetivo: Codificar un ejemplo de TypeScript que utilice clases, tipado e interfaces.

Instrucciones:

1. Transcribir el siguiente código a TypeScript:

```
var nombre;
nombre = "Miguelo";
var edad;
edad = 30;
var PERSONAJE = { nombre: nombre, edad: edad };
```

2. Usar **let** y **const**.
3. Usar **tipado estricto**.
4. Crear una **Interface** en TypeScript.

Resolución:

```
interface Personaje {
  nombre: string;
  edad: number;
}

let nombre: string = 'Miguelo';
let edad: number = 30;

const PERSONAJE: Personaje = {
  nombre: nombre,
  edad: edad
};
```

Actividad 3. Clases e interfaces

Duración: 10 minutos

Objetivo: Crear interfaces y datos que se utilicen para mostrarlos en un componente.

Instrucciones:

1. Crear 2 archivos **persona.ts** e **informacion-contacto.ts**.

2. En **persona.ts**, crear una **interface** con la siguiente estructura:
    - **nombre**: string
    - **apellidos**: string
    - **edad**: number
    - **direccion**: string
3. En **informacion-contacto.ts**, importar la **interface** y crear un objeto que cargue los datos de la interface y los muestre en la consola.

Resolución:

```
// persona.ts
export interface Persona {
  nombre: string;
  apellidos: string;
  edad: number;
  direccion: string;
}
```

```
// informacion-contacto.ts
import { Persona } from './persona';

const contacto: Persona = {
  nombre: 'Pedro',
  apellidos: 'González López',
  edad: 28,
  direccion: 'Calle Falsa 123, Ciudad Angular'
};

console.log('Información de contacto:', contacto);
```

Actividad 4. Componente & Directivas con *nglf y ngFor*

Duración: 20 minutos

Objetivo: Crear un componente que utilice las directivas ***nglf** y ***ngFor**.

Instrucciones:

1. Usar Angular CLI para crear un nuevo componente.
2. Usar ***nglf** para renderizar condicionalmente una vista.
3. Usar ***ngFor** para recorrer una lista.

Resolución:

```
// ejemplo.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-ejemplo',
  templateUrl: './ejemplo.component.html',
  styleUrls: ['./ejemplo.component.css']
})
export class EjemploComponent {
  mostrar: boolean = true;
  elementos: string[] = ['Elemento 1', 'Elemento 2', 'Elemento 3'];
}
```

```
<!-- ejemplo.component.html -->
<div *ngIf="mostrar">
  <h1>Directiva *ngIf activa</h1>
</div>

<ul>
  <li *ngFor="let elemento of elementos">{{ elemento }}</li>
</ul>
```

Actividad 5. Componente con ngClass y Pipes

Duración: 15 minutos

Objetivo: Crear un componente que utilice la directiva **ngClass** y un **Pipe**.

Instrucciones:

1. Crear un componente con Angular CLI.
2. Usar **ngClass** para cambiar el color de la fuente.
3. Usar un **Pipe** para convertir un texto a mayúsculas.

Resolución:

```
// pipes-ejemplo.component.ts
import { Component } from '@angular/core';
```

```
@Component({
  selector: 'app-pipes-ejemplo',
  templateUrl: './pipes-ejemplo.component.html',
  styleUrls: ['./pipes-ejemplo.component.css']
})
export class PipesEjemploComponent {
  texto: string = 'texto en minúsculas';
  activarClase: boolean = true;
}
```

```
<!-- pipes-ejemplo.component.html -->
<p [ngClass]="{'resaltado': activarClase}">
  Este texto cambiará su estilo con ngClass
</p>

<p>{{ texto | uppercase }}</p>
```

```
/* pipes-ejemplo.component.css */
.resaltado {
  color: red;
  font-weight: bold;
}
```

Actividad 6. Actividad Colaborativa: Find the Bug

Duración: 15 minutos

Objetivo: Identificar y corregir errores en código con errores relacionados con **interpolación**, **Property Binding** y **directivas estructurales**.

Instrucciones:

1. Se presenta un fragmento de código con errores.
2. Los estudiantes deben identificar los errores.
3. Los estudiantes proponen la corrección.

Ejemplo de código con errores:

```
// Código con errores
export class ComponenteErroneo {
  nombre = 'Juan';
  <p>{{ nombre }}</p> // Error: no se puede usar HTML dentro de
TypeScript
}
```

Corrección:

```
// componente-correcto.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-componente-correcto',
  templateUrl: './componente-correcto.component.html',
  styleUrls: ['./componente-correcto.component.css']
})
export class ComponenteCorrectoComponent {
  nombre: string = 'Juan';
}
```

```
<!-- componente-correcto.component.html -->

<p>{{ nombre }}</p>
```