Contenido Procesado

1. angular: 32 veces
2. typescript: 20 veces
3. javascript: 10 veces
4. binding: 10 veces
5. propiedades: 9 veces
6. private: 9 veces
7. public: 9 veces
8. export: 8 veces
9. tipo: 7 veces
10. valor: 7 veces

DWEC05.- angular JS y POO en typescript

En esta UD veremos **los principios de programación orientada a objetos en typescript** y empezaremos a usar el **framework angular JS**, con la intención de desarrollar nuestra primera web SPA bajo los estándares de un framework utilizado actualmente en el mercado laboral.

Iremos desarrollando todo esto a lo largo de la teoría y vídeos de la UD, pero además de esto me parece interesante que consultéis los enlaces que os dejo para poder profundizar más en el tema, ya que por las limitaciones temporales del módulo no es posible hacer hincapié a un nivel más profundo en todo lo que se comenta. Por tanto me gustaría facilitaros algunas referencias para que podáis trabajar algunos aspectos por vuestra cuenta.

Curso angular JS y POO con typescript

Enlace a GitHub con ejercicios del curso

1.- POO en typescript

¿Qué es? Es un superset de Microsoft con extensión ".ts". Es un 85% javascript al que se le han añadido todos los últimos estándares ECMASCRIPT. Lo que hace TS es transpìlar el código TS a JS para que funcione en la mayoría de los navegadores.

Aparte de esto tiene un tipado de datos fuerte y tiene opciones avanzadas para POO

Recomendación

POO ejercicios introductorios

2.- Primeros pasos con typescript

Qué Es typescript

javascript es uno de los lenguajes más populares, en parte porque ha evolucionado y mejorado a pasos agigantados en los últimos años.

Sin embargo, javascript en algún punto fue un lenguaje que presentaba muchos problemas para bases de código grandes, aplicaciones de gran escala y proyectos con muchos años de desarrollo.

En 2012 en javascript no habían clases, ni módulos, el ecosistema carecía de herramientas que optimizaran el flujo de desarrollo, derivado precisamente por las carencias del lenguaje mismo.

2012 fue el año en que typescript apareció (luego de 2 años de desarrollo), una solución de Microsoft para el desarrollo de aplicaciones con javascript a gran escala, para ellos y para sus clientes. Steve Lucco y un equipo de más de 50 personas que incluía a Anders Hejlsberg, Lead Architect de C# y creador de Delphi y Turbo Pascal desarrollaron typescript en Microsoft, un proyecto que originalmente se conoció como Strada.

Originalmente, productos como Bing y Office 365 despertaron en Microsoft la necesidad de una mejora a javascript que permitiera construir productos escalables.

typescript es la solución a muchos de los problemas de javascript, está pensado para el desarrollo de aplicaciones robustas, implementando características en el lenguaje que nos permitan desarrollar herramientas más avanzadas para el desarrollo de aplicaciones.

SUPERSET DE javascript

typescript es un superset de javascript. Decimos que una tecnología es un superset de un lenguaje de programación, cuando puede ejecutar programas de la tecnología, typescript en este caso, y del lenguaje del que es el superset, javascript en este mismo ejemplo. En resumen, esto significa que los programas de javascript son programas válidos de typescript, a pesar de que typescript sea otro lenguaje de programación.

Esta decisión fue tomada en Microsoft bajo la promesa de que las futuras versiones de Ecmascript traerían adiciones y mejoras interesantes a javascript, esto significa que typescript se mantiene a la vanguardia con las mejoras de javascript.

Además, esto permite que uno pueda integrar typescript en proyectos existentes de javascript sin tener que reimplementar todo el código del proyecto en typescript, de hecho, es común que existan proyectos que introduzcan tanto typescript como javascript.

Por si fuera poco, uno de los beneficios adicionales de esta característica del lenguaje, es que pone a disposición el enorme ecosistema de librerías y frameworks que existen para javascript. Con typescript podemos desarrollas aplicaciones con React, Vue o angular que es el que estudiaremos en esta UD

2.1.- TS tipo de datos Básicos

En este apartado mencionaremos con ejemplos en código los tipos de datos primitivos que os encontraréis en typescript

```
/ tipo de original
let bool: boolean = true
let num: number = 123
let str: string = 'abc'


//Estructuras de datos
let arr1: number[] = [1,2,3]
let arr2: Array<number | string> = [1,2,3,'4']



let tuple: [number,string] = [0,'1']
//Función
let add = (x:number,y:number)  => x+y
let compute: (x:number,y:number) =>number
compute = (a,b) => a+b
//Objeto
let obj: {x:number,y:number} = {x:1,y:2}ftype
obj.x = 3
// el símbolo tiene un valor único
let s1: symbol = Symbol()
let s2 = Symbol()
console.log(s1 === s2) //false
// any es equivalente a la escritura de js
let x: any
x=123
x=[]
x=()=>{}
// 1. Lanzar una excepción
let error = ()=> {
throw new Error("error")
}
```

2.2.- TS tipo Avanzados

En este apartado exploraremos de una manera superficial los recursos de programación más avanzados que podéis encontrar en typescript

2.2.1. - Enum

```
// La enumeración

enum Role {

 Reporter = 1,

 Developer,

 Maintainer,

 Owner,

 Guest

// Si no hay un valor predeterminado, entonces el valor del índice comienza desde 0, de lo contrario aumenta desde el valor predeterminado hacia abajo

// console.log(Role)

// enumeración de cadenas

enum Message {

    Success = 'Felicitaciones, lo logró',

    Fail = 'Lo siento, falló'

// Números de enumeración heterogéneos y cadenas mezcladas

enum Answer {

 N,

 Y = 'yes'
```

2.2.2. - Interfaz

SI estoy trabajando con una clase, puedo implementar interfaces, para que todas las clases que implementan dicha interfaz tengan definidos los mismos métodos, pero sin implementar

```
interface Lib {

 (): void;

 version: string;

 doSomething(): void;

function getLib(){

 let lib: Lib = (() => ()) como Lib; // escribe aserción

 lib.version ='1.0';

 lib.doSomething =()=> {}

let lib1 = getLib;

lib1();

interface StringArray {

 [index: number]: string

let chars: StringArray = ['A','B']
```

2.2.3. - Función

Tened en cuenta que aquí os doy muchas opciones, pero lo más común es usar la primera forma:

```
// Los siguientes métodos de escritura son consistentes en expresión
function add1(x: number, y: number){
 return x+y
}
let add2: (x: number, y: number) => number
type add3 = (x: number, y: number) => number
interface add4 {
 (x: number, y: number): number
}

// Parámetros opcionales: agregue un signo de interrogación y debe colocarse después de los parámetros requeridos
function add5(x: number, y?: number){
 return y? x=y: x
}
add5(1)
// Función con valor inicial
function add6(x: number, y=0,z:number,q=1){
 return x+y+z+q
}
add6(1,undefined,3)
function add7(x:number,...rest:number[]){
```

```
 return x+rest.reduce((pre,cur)=>pre+cur)
}
console.log(add7(1,2,3,4,5))

// La sobrecarga de funciones se usa a menudo para lograr funciones similares pero se manejan diferentes tipos de datos
function add8(...rest: number[]): number;
function add8(...rest: string[]): string;
function add8(...rest: any[]): any{
 let first = rest[0]
 if(typeof first === 'string'){
 return rest.join('')
 }
 if(typeof first === 'number'){
 return rest.reduce((pre,cur)=>pre+cur)
 }
}
console.log(add8(1,2,3))
console.log(add8('a','b','c','d'))
```

2.2.4. - Objetos

```
// clase - debe llamarse igual que el fichero

class Coche{


    // propiedades - pueden ser public/private/protected, en función a desde donde yo quiera acceder

    private color: string;

    private modelo: string;

    private marca: string;

    private precio: number;

    // constructor

    constructor(color,modelo,marca, precio){

        this.color = color;

        this.modelo=modelo;

        this.marca=marca;

        this.precio = precio;

    }



    // propiedades

    public cambiarColor(color){

        this.color;

    public getColor(){

        return this.color;

    //metodos

    public acelerar(){

        console.log("Coche acelerando a 100Km/h")




// crear coche

var coche = new Coche("","","",0);

// usar propiedades

coche.cambiarColor("rojo");

let c : string = coche.getColor();
```

2.2.5. - Main, Import y export

- Main

```
// podemos crear un archivo Main para cuando se inicialice nuestro programa

// y desde aquí ir llamando a los diferentes modulos

class Main{

 constructor(){
```

```
  console.log("Aplicación Cargada!")
 }
var main = new Main();
```

- Import/export

Para poder usar por ejemplo la clase Coche desde el main tendré que hacer dos cosas. Poner la etiqueta export en la cabecera de la clase que quiero exportar, y la etiqueta import nombre de la clase en fichero donde yo lo quiera usar, el main en este caso

```
export class Coche{...
import {Coche} from ‘./je’...
```

2.2.6. - Herencia

Cuando estoy trabajando en una clase, puedo heredar atributos y métodos de otra, utilizando la palabra reservada extends. Y además implementar métodos concretos para dicha clase. En este caso trabajaremos con la interfaz anterior y las clases Camiseta y Sudadera

```
class Camiseta implements CamisetaBase{
    // Atributo
    private marca:string;
    private precio:number;
    constructor(color:string, modelo:string, marca:string, precio:number){
        this.modelo = modelo;
        this.marca = marca;
    setModelo(modelo: string) {
    getModelo(): string {
        return this.modelo;
    setMarca(marca: string) {
    getMarca(): string {
        return this.marca;
    setPrecio(precio: number) {
        this.precio=precio;
    getPrecio(): number {
        return this.precio;;
    // propiedades
    public setColor(color:string){
        this.color=color;
    public getColor():string{
       return this.color;
    // Metodos
class Sudadera extends Camiseta{
    private capucha:boolean;
    setCapucha(capucha:boolean){
        this.capucha = capucha;
```

2.2.7. - Decoradores

Un decorador es un patrón de diseño que nos permite mediante a través de metadatos, cambiar la funcionalidad de una clase en función de los parámetros que le pasemos

Primero lo tengo que definir. Siguiendo con nuestro ejemplo de la camiseta y sudadera creare el decorador estampar.

```
function estampar(logo:string){
    return function(target: Function){
        target.prototype.estampacion = function():void{
            console.log("camiseta estampada " + logo);
            }
        }
```

Ahora, si yo añado este decorador a la clase con la siguiente sintaxis, la clase implementará el nuevo método estampar que lo llamaría cómo si fuera un método implementado de manera usual, y recibe por parámetro lo que yo haya implementado en el decorador

```
@estampar("Gucci")

class Camiseta implements CamisetaBase{ ….
```

3.- Introducción al Framework angular

angular es un framework opensource desarrollado por Google para facilitar la creación y programación de aplicaciones web de una sola página, las webs SPA (Single Page Application).

angular separa completamente el frontend y el backend en la aplicación, evita escribir código repetitivo y mantiene todo más ordenado gracias a su patrón MVC (Modelo-Vista-Controlador) asegurando los desarrollos con rapidez, a la vez que posibilita modificaciones y actualizaciones.

En una web SPA aunque la velocidad de carga puede resultar un poco lenta la primera vez que se abre, navegar después es totalmente instantáneo, ya que se ha cargado toda la página de golpe.

Solamente es una ruta la que se tiene que enviar al servidor, y angular lo que hace 'por debajo' es cambiar la vista al navegar para que dé la apariencia de una web normal, pero de forma más dinámica.

Entre otras ventajas, este framework es modular y escalable adaptándose a nuestras necesidades y al estar basado en el estándar de componentes web, y con un conjunto de interfaz de programación de aplicaciones (API) permite crear nuevas etiquetas HTML personalizadas que pueden reutilizarse.

El lenguaje principal de programación de angular es typescript, y así toda la sintaxis y el modo de hacer las cosas en el código es el mismo, lo que añade coherencia y consistencia a la información, permitiendo por ejemplo, la incorporación de nuevos programadores, en caso de ser necesarios, ya que pueden continuar su trabajo sin excesiva dificultad.

Como ya se ha indicado, las plantillas de angular almacenan por separado el código de la interfaz del usuario (front-end) y el de la lógica de negocio (back-end), que entre otros beneficios permite utilizar mejor otras herramientas anteriormente existentes.

Y por si fuera poco, los principales editores y entornos de desarrollo integrado (IDEs) ofrecen ya extensiones para poder trabajar con este framework con mayor comodidad.

Por su programación reactiva, la vista se actualiza automáticamente tras realizar los cambios. Además angular dispone de asistente por línea de comandos para poder crear proyectos base y también se integra bien con herramientas de testing y con Ionic, lo que facilita la creación de web-responsive, es decir, adaptadas a móviles.

Este aspecto cada día adquiere mayor importancia tanto por el creciente uso de estos dispositivos para acceder a internet como por la penalización que Google realiza de aquellas páginas que no facilitan su visita en cualquier dispositivo.

3.1. - Instalar angular

Antes de comenzar, tienes que tener VSCode y Node.JS instalados.

En la página oficial de angular tenemos una pequeña guia de cómo instalarlo y cuales son los primeros pasos. Además podemos consultar su directorio de GitHub, en el que hay información muy interesante sobre el tipo de cosas que podemos hacer etc.

Para instalarlo desde VSCode abre la terminal. Como ya tenemos NodeJS instalado podemos ejecutar comandos npm para instalar angular.
Para la instalación de angular debemos ejecutar el siguiente comando:

```
npm install -g @angular/cli
```

Este comando instala angular, el -g sirve para agregarlo de manera global a Windows.

3.2. - Crear un Proyecto

Ya con angular instalado podemos ir a la ruta donde queremos tener nuestro primer proyecto y allí debemos ejecutar el siguiente comando.

```
ng new nombre-proyecto
```

Este comando tarda algunos minutos y debe crear automáticamente el directorio que se ve en la siguiente imagen.

Después nos colocaremos sobre el directorio del proyecto, y ejecutaremos el siguiente comando para compilar la aplicación y poder empezar a desarrollar:

```
ng serve
```

Una vez compilado, veremos lo siguiente:

Por un lado nos dice que el servicio Live de angular se estará ejecutando en el puerto Localhost 4200, es decir si abrimos un navegador en

esa dirección veremos la página principal que angular genera por defecto. Cabe destacar, que angular es de tipo SPA y esta pagina no se recargará ni se refrescara constantemente, simplemente ira cambiando dinámicamente los contenidos a medida que interactuamos con ella.

Y por otro lado, si abrimos nuestro proyecto angular en VSCode, veremos el siguiente scaffolding de angular por defecto

3.3. - Componentes

Un componente simplemente es una de las partes de la interfaz de nuestra aplicación, y la lógica que lo controla internamente. Por ejemplo; el header de la página, un calendario, un sidebar, un formulario de contacto…

Un componente en angular es un elemento que está compuesto por:

- Un archivo que será nuestro Template (app.component.html), el cual es nuestro HTML, que es el que se va a visualizar en la interfaz de usuario, la vista o en términos más simples lo que vas a ver en la página.

Un archivo que será nuestro Template (app.component.html), el cual es nuestro HTML, que es el que se va a visualizar en la interfaz de usuario, la vista o en términos más simples lo que vas a ver en la página.

- Un archivo de lógica, la cual es la que pondremos en un archivo .ts (como por ejemplo app.component.ts), ese archivo debe incluir una clase y esta es la que va a contener las propiedades que se van a usar en la vista (HTML) y los métodos que será las acciones que se ejecutarán en la vista. En este archivo de lógica también se incluye una metadata, que es definida con un decorador, que identifica a angular como un componente.

Un archivo de lógica, la cual es la que pondremos en un archivo .ts (como por ejemplo app.component.ts), ese archivo debe incluir una clase y esta es la que va a contener las propiedades que se van a usar en la vista (HTML) y los métodos que será las acciones que se ejecutarán en la vista. En este archivo de lógica también se incluye una metadata, que es definida con un decorador, que identifica a angular como un componente.

Crear un componente:

```
ng generate component <name>
```

3.4. - Modelos de Datos

Un modelo en angular es una clase que representa un objeto con propiedades que permite la utilización de la misma desde clases externas, evitando la reiteración de código. A lo largo de esta entrada veremos cómo construir un modelo y cómo implementarla en componentes.

Estas clases se colocan dentro de src/app en una carpeta llamada models.

El nombre de estos ficheros se escribe en minúscula:

nombrefichero.ts. Como mencionamos previamente, un modelo es una clase que define un objeto con sus respectivas características, por ende, su sintáxsis es la siguiente:

```
export class NombreClase{

  constructor(

  public propiedad : tipo;

  public propiedad: tipo;

  ){}
```

La sintaxis previa declara las propiedades dentro del constructor, lo cual resulta una gran ventaja para el programador. También existe otra forma un poco más larga que declara las propiedades fuera del constructor y las inicializa dentro del mismo en base a los parámetros recibidos:

```
    constructor(valor1, valor2){

     this.propiedad : valor1;

     this.propiedad: valor2;
```

Para implementar un modelo debemos importarlo en el componente que deseemos utilizarlo. Su sintaxis es la siguiente:

```
Import { NombreModelo } from 'ruta/models/modelo.ts'
```

3.5. - Directivas

Las Directivas extienden la funcionalidad del HTML usando para ello una nueva sintaxis. Con ella podemos usar lógica que será ejecutada en el DOM (Document Object Model).

Cada Directiva que usamos tiene un nombre, y determina donde puede ser usada, sea en un elemento, atributo, componente o clase.

- ngIf

  ngIf

  Nos permite incluir condicionales de lógica en nuestro código, como por ejemplo evaluar sentencias, hacer comparaciones, mostrar u ocultar secciones de código, y entre las muchas condiciones que deseemos crear, para que se renderice nuestro HTML, cumpliendo la sentencia a evaluar. Con el *ngIf, podemos evaluar sentencias con un simple If, podemos evaluar el else, para que no cumpliéndose la primera condición que se evalúa nuestro código ejecute otra acción en el caso contrario y podemos además incluir el then, para que cumpliendose la condición afirmativa (if), podamos añadir más flexibilidad a nuestro código incluyéndole un camino afirmativo adicional.

- ngFor

  ngFor

  Permite ejecutar bucles, los bucles son los que conocemos en lógica de programación como: for, while, foreach, etc. Con esta directiva estructural podemos evaluar de acuerdo a nuestra condición n veces.

- ngSwitch

  ngSwitch

  Esta directiva es similar al *ngIf, y es como el switch en lógica de programación. En esta directiva se pueden crear los diferentes casos que deseamos evaluar y cuando se cumple la condición esperada, oculta/muestra el HTML. Nos permite mantener nuestro código más limpio, si necesitamos evaluar varias sentencias.

3.6. - Enlace de datos - Data binding

Hasta ahora, hemos estado trabajando con comunicaciones/sincronizaciones (bindings) unidireccionales, es decir, de una sola dirección, los conocidos como One Way Data binding.

En los que nos traspasábamos/comunicábamos/sincronizábamos un objeto o una variable de nuestra Modelo, es decir, la lógica de negocio del componente situado en nuestro código de typescript hacía el template (el HTML) o viceversa.

En esta gráfica podemos ver más detalladamente hacía que dirección trabaja cada uno de los bindings. Y podemos ver que hay bindings trabajan del componente hacía la vista todos los que hemos explicado a excepción del Event binding que trabaja de la vista hacía el componente.

En muchas ocasiones nos encontramos que este tipo de bindings los que se denominan bidireccionales son los que más necesitamos en la programación del día a día. Para ello angular nos provee una sintaxis más compacta que indica que este binding es BiDireccional combinando corchetes y paréntesis [()]. Para ello el primer paso que tenemos que realizar es instalar a nivel de la aplicación el FormsModule que es el módulo que nos ayuda a gestionar formularios.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule} from "@angular/forms"
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { C1Component } from './c1/c1.component';
@NgModule({
 declarations: [
   AppComponent,
   C1Component
 ],
 imports: [
   BrowserModule,
   AppRoutingModule,
   FormsModule
 providers: [],
 bootstrap: [AppComponent]
})
```

```
export class AppModule { }
```

Una vez configurado el Forms módulo podemos actualizar nuestra plantilla y usar la directiva de angular ngModel para realizar un binding bidireccional como teníamos en el ejemplo anterior pero mucho más compacto con corchetes y paréntesis.

```
<input type="text" name="nombre" [(ngModel)]="nombre" />
  {{nombre}}
```

De esta manera tendremos una caja de texto que contiene el nombre que nosotros necesitamos . Si cambiamos el valor de la caja automáticamente nos cambiará el valor del texto que tenemos a nivel de la interpolación, el binding será bidireccional

Acabamos de usar angular la directiva ngModel para construir nuestro binding .

Muchas veces la gente cuando comienza con angular se suele equivocar mucho a la hora de realizar los binding bidireccionales y que no sabe cómo poner los corchetes o los paréntesis. Para ello la gente de angular decidió definir una regla de nemotecnia y llamo a este tipo de binding "banana in box" porque se trata de una banana dentro de una caja. Así es imposible confundirse.

angular ngModel y objetos

Recordemos que los bindings no solo se pueden aplicar a tipos básicos sino que pueden ser aplicados a objetos complejos que provengan de clases y asignar las propiedad que nosotros deseemos. Por ejemplo si disponemos de la clase Persona:

```
export class Persona {
   nombre:string;
   edad:number;
```

Podemos usar angular ngModel y realizar un two way data binding sobre sus propiedades:

```
import { Component, OnInit } from '@angular/core';
import { Persona } from '../persona';
@Component({
  selector: 'app-c1',
  templateUrl: './c1.component.html',
  styleUrls: ['./c1.component.css']
export class C1Component implements OnInit {
  persona:Persona;
  constructor() {
   this.persona= new Persona();
   this.persona.nombre="gema";
   this.persona.edad=20;
  }
  ngOnInit() {
```

Acabamos de usar la clase Persona a nivel de componente nos falta ver su información a nivel de la propia plantilla:

```
<input type="text" name="nombre" [(ngModel)]="persona.nombre" />
<input type="text" name="edad" [(ngModel)]="persona.edad" />
{{persona.nombre}}
{{persona.edad}}
```

Si vemos el resultado en el navegador nos cargará la información sobre el objeto:

Si cambiamos la información automaticamente los datos se actualizan en las interpolaciones.

Aprender a usar binding bidireccionales o two way databindings con angular apoyándonos en la sintaxis de banana in box [()] . Esto nos facilitará sobre manera el trabajo con componentes en el día a día. Sobre todo cuando trabajamos con estructuras de objetos complejas que

demandan un mapeo de propiedades no trivial como acabamos de ver en este último caso de las personas.