Angular

**INSTALAR ANGULAR/CLI**

Necesitamos instalar **angular/cli** con el comando:

**Npm install –g @angular/cli**

Y luego ejecutar el comando:

**ng versión**

**SEC-1: BASES DE TYPESCRIPT**

**TIPOS BÁSICOS Y CONCEPTOS GENERALES**

Podemos especificarle a la variable que tipo de datos es el valor de la misma:

let nombre: string = "joaquin";

Como también podemos especificar que la variable pueda recibir dos tipos de datos:

let numberAndString: number | string = 95;

numberAndString = "hola";

**OBJETOS, ARREGLOS E INTERFACES**

Debemos especificarle al array que tipos de datos puede recibir:

let habilidades: (boolean | string | number)[] = ["Bash", "Counter", 100]

habilidades.push(true);

Para especificar los tipos de datos que recibe un objeto utilizamos una **interface**, esta interface es una estructura base de lo que tiene que contener el objeto a crear. Para agregar **propiedades** **opcionales** hacemos uso del **signo de interrogación (?)**.

interface Personaje {

    nombre: string,

    hp: number,

    habilidades: string[],

    puebloNatal?: string

}

const personaje: Personaje = {

    nombre: "joaquin",

    hp: 24,

    habilidades: ["Bash", "Counter", "Healing"]

}

personaje.puebloNatal = "Pueblo Paleta";

console.table( personaje );

**FUNCIONES BÁSICAS**

A las funciones le podemos especificar qué tipo de datos reciben y también que tipo de dato esperamos que sea devuelto:

function sumar (a: number, b: number): number {

    return a + b;

}

const resultado = sumar(10, 20);

console.log(resultado)//30

**Arrow function:**

const sumarFlecha = (a: number, b: number):number => {

   return a + b

}

Podemos pasar argumentos opcionales y por default. El orden para colocar bien los argumentos son: requeridos, opcionales y por default.

function multiplicar (a: number, b?: number, c: number = 2) {

    return a \* c

}

const resultadoMultiplicacion = multiplicar(10, 0, 10)

console.log(resultadoMultiplicacion)//100

**FUNCIONES CON OBJETOS COMO ARGUMENTOS**

Si en una function no quiero retornar nada utilizo **void**.

interface PersonajeLOR {

    nombre: string,

    pv: number,

    mostrarPV: () => void

}

function curar (personaje: PersonajeLOR, curarX: number): void {

    personaje.pv += curarX;

    console.log(personaje)

}

const nuevoPersonaje: PersonajeLOR = {

    nombre: "Frodo",

    pv: 50,

    mostrarPV() {

        console.log("Puntos de vida: ", this.pv)

    }

}

curar(nuevoPersonaje, 20);

nuevoPersonaje.mostrarPV();

**TAREA SOBRE OBJETOS E INTERFACES**

interface SuperHero {

    nombre: string,

    edad: number,

    direccion: Direccion,

    mostrarDireccion: () => string

}

interface Direccion {

    calle: string,

    pais: string,

    ciudad: string

}

const superHero = {

    nombre: "Spiderman",

    edad: 30,

    direccion: {

        calle: "Main St",

        pais: "USA",

        ciudad: "NY"

    },

    mostrarDireccion () {

        return `${this.nombre}, ${this.direccion.ciudad}, ${this.direccion.pais}`

    }

}

const direccion = superHero.mostrarDireccion();

console.log(direccion);

**DESTRUCTURACIÓN DE OBJETOS**

interface Reproductor {

    volumen: number,

    segundo: number,

    cancion: string,

    detalles: Detalles,

}

interface Detalles {

    autor: string,

    anio: number

}

const reporductor: Reproductor = {

    volumen: 90,

    segundo: 36,

    cancion: "Mess",

    detalles: {

        autor: "Ed Sheeran",

        anio: 2015

    }

}

const autor = "Fulano";

const {volumen, segundo, cancion, detalles} = reporductor;

const {autor: autorDetalle} = detalles

console.log(`El volumen actual es: ${volumen}`);

console.log(`El segundo actual es: ${segundo}`);

console.log(`La canción actual es: ${cancion}`);

console.log(`El autor actual es: ${autorDetalle}`);

**DESESTRUCTURACIÓN DE ARRAYS**

const dbz: string[] = ["Goku", "Vegeta", "Trunks"];

const [goku, vegeta, trunks] = dbz

console.log(`Personaje 1: ${goku}`);

console.log(`Personaje 2: ${vegeta}`);

console.log(`Personaje 3: ${trunks}`);

**DESESTRUCTURACIÓN DE ARGUMENTOS**

interface Producto {

    desc: string,

    precio: number

}

const telefono: Producto = {

    desc: "Nokia A1",

    precio: 150

}

const tablet: Producto = {

    desc: "iPad Air",

    precio: 350

}

function calculaISV (productos: Producto[]):[number, number] {

    let total = 0;

    productos.forEach(({precio}) => {

        total += precio;

    })

    return [total, total \* 0.15];

}

const articulos = [telefono, tablet];

const [total, isv] = calculaISV(articulos);

console.log(total, isv)

**IMPORTACIONES Y EXPORTACIONES**

Utilizamos la palabra clave **export** en la interface que vamos a exportar, y luego en el archivo donde queremos importar esa interface utilizamos la palabra clave **import** Interface **from** “ruta”. (Podemos apretar ctrl + . y nos aparecerá la importación a realizar).

**Archivo 06:**

export interface Producto {

    desc: string,

    precio: number

}

export function calculaISV (productos: Producto[]):[number, number] {

    let total = 0;

    productos.forEach(({precio}) => {

        total += precio;

    })

    return [total, total \* 0.15];

}

**Archivo 07:**

import { calculaISV, Producto } from "./06-desest-argumentos";

const carritoCompras: Producto[] = [

    {

        desc: "Iphone 13",

        precio:1300

    },

    {

        desc: "Iphone 12",

        precio: 1000

    }

];

const [total, isv] = calculaISV(carritoCompras);

console.log(`Total: ${total}`);

console.log(`ISV: ${isv}`);

**CLASES BÁSICAS**

**Private:** visible solo dentro de la clase. Necesitamos un constructor para utilizarla.

**Public:** se puede ver desde afuera de la clase.

**Static:** podemos acceder al valor de la propiedad sin crear una instancia de la clase.

class Heroe {

    alterEgo: string;

    edad: number;

    nombreReal:number;

}

const ironMan = new Heroe();

**CONSTRUCTOR DE UNA CLASE**

El **constructor** no es más que una función que se va a llamar cuando creo una instancia de la clase, es decir, cuando escribir **new Heroe()**.

class Heroe {

    // alterEgo: string;

    // edad: number;

    // nombreReal:number;

    // constructor(alterEgo: string){

    //     this.alterEgo = alterEgo;

    // }

    constructor (

        public alterEgo: string,

        public edad: number,

        public nombreReal:string,

    ){}

}

const ironMan = new Heroe("Iron Man", 40, "Tony");

console.log(ironMan)

**EXTENDER UNA CLASE**

En la clase que se extiende de otra necesitamos pasarle el **super()** con los argumentos que le pasamos en la clase Padre.

class PersonaNormal {

    constructor(

        public nombre: string,

        public direccion: string

    ){}

}

class Heroe extends PersonaNormal {

    constructor (

        public alterEgo: string,

        public edad: number,

        public nombreReal:string,

    ){

        super(nombreReal, "New York, USA");

    }

}

const ironMan = new Heroe("Iron Man", 40, "Tony");

console.log(ironMan)

**GENÉRICOS**

Se utiliza para especificar que puede llegar a pasarse cualquier tipo de valor en un argumento. Utilizamos **<T>** para decir que esperamos cualquier genérico. Es como un comodín.

function queTipoSoy<T>(argumento: T) {

    return argumento;

}

let soyString = queTipoSoy("Joaquin");

let soyNumber = queTipoSoy(24);

let soyArray = queTipoSoy([1,2,3]);

let soyExplícito = queTipoSoy<number>(100);

**DECORADORES DE CLASES**

Sirven para cambiar las clases en el momento en que son definidas.

Para utilizarlos debemos ir al archivo **tsconfig.json** y en donde dice “Experimental Options” hay que descomentar la línea **“experimentalDecorators”:true**.

function classDecorator<T extends {new (...args: any[]): {} }>(

    constructor: T

) {

    return class extends constructor {

        newProperty = "new property";

        hello = "override";

    };

}

@classDecorator

class MiSuperClase{

    public miPropiedad: string = "AB22";

    imprimir(){

        console.log("Hello World");

    }

}

console.log(MiSuperClase);

**ENCADENAMIENTO OPCIONAL**

interface Pasajero {

    nombre: string;

    hijos?: string[];

}

const pasajero1: Pasajero = {

    nombre: "Joaquín",

}

const pasajero2: Pasajero = {

    nombre: "Lionel",

    hijos: ["Thiago", "Mateo", "Ciro"],

}

function imprimeHijos (pasajero: Pasajero):void {

    const cantidadHijos = pasajero.hijos?.length || 0;

    console.log(cantidadHijos);

}

imprimeHijos(pasajero1);

imprimeHijos(pasajero2);

**SEC-4: INTRODUCCIÓN a ANGULAR**

**CREAR UN PROYECTO EN ANGULAR**

Dentro de una carpeta correr el comando **ng new nombreDelProyecto**.

Luego ubicarnos dentro de la carpeta del proyecto y correr el comando **ng serve -o**.

**EXPLICACIÓN DE LOS ARCHIVOS DENTRO DEL SRC**

Ubicarnos en src/app/app.component.html y seleccionar todo con ctrl + a, y borremos todo. Pongamos una etiqueta <h1> con “Hola Mundo”.

**APP COMPONENT**

En el archivo src/app/app.component.ts vamos a tener el **@Component** el cual va a recibir:

* **Selector:** con el “app-root” que es el root que se encuentra en el index.html.
* **TemplateUrl:** la ruta que especificamos para llegar al archivo app.component.html.

**CREANDO UN COMPONENTE**

Crear un archivo **nombre-que-queramos.component.ts**, hacer un import de:

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

Y crear el @Component({}):

@Component({

    selector: "app-contador",

    template: `

    <h1>{{ titulo }}</h1>

    <h3>La base es: <strong>{{ base }}</strong></h3>

    <button (click)="acumular(base)">+ {{ base }}</button>

    <span>{{ numero }}</span>

    <button (click)="acumular(-base)">- {{ base }}</button>

    `

})

En este caso usamos un **template**, pero se suele usar un **templateUrl**.

Luego crear la **class**:

export class ContadorComponent {

    titulo: string = 'Contador App';

    numero: number = 10;

    base: number = 5;

    acumular(valor: number) {

        this.numero += valor;

    }

}

A continuación ir al archivo app.modules.ts e importar el componente creado:

import { ContadorComponent } from './contador.component';

Y en el **@NgModule** en “declarations” colocar el nombre del componente:

@NgModule({

  declarations: [

    AppComponent,

    ContadorComponent

  ],

  imports: [

    BrowserModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

Por último en el archivo app.component.html colocar el componente creado:

<app-contador></app-contador>

**CREAR COMPONENTE DE FORMA AUTOMÁTICA**

En la consola correr el comando: **ng g c laRutaDondeQueremosCrearElComponente**

**DIRECTIVA \*ngFor**

Utilizamos la directa \*ngFor para poder loopear un array con el ciclo FOR.

**Html:**

<h1>Listado de Heroes:</h1>

<div>

    <h3>Heroe Borrado:</h3>

    <span>{{ heroeBorrado }}</span>

</div>

<button (click)="deleteHeroe()">

    Borrar

</button>

<ul>

    <li \*ngFor="let heroe of heroes; let i = index">

        {{ i + 1 }} - {{ heroe }}

    </li>

</ul>

**Component.ts:**

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

@Component({

  selector: 'app-listado',

  templateUrl: './listado.component.html',

})

export class ListadoComponent {

  heroes: string[] = ["SpiderMan", "IronMan", "Hulk", "Thor"];

  heroeBorrado: string = "";

  deleteHeroe(){

    this.heroeBorrado= this.heroes.pop() || "";

  }

}

**DIRECTIVA \*ngIf**

Se utilizan para hacer validaciones if:

<div \*ngIf="heroeBorrado.length > 0 ? true : false">

    <h3>Heroe Borrado: <small>{{ heroeBorrado }}</small></h3>

</div>

**Ng-Template y el ngIf-Else**

<div \*ngIf="heroeBorrado; else noBorrado">

    <h3>Heroe Borrado: <small>{{ heroeBorrado }}</small></h3>

</div>

<ng-template #noBorrado>

    <h3>No borraste nada aún</h3>

</ng-template>

**MÓDULOS**

Dentro de un archivo **module** necesitamos importar el NgModule, y utilizarlo:

import { CommonModule } from "@angular/common";

import { NgModule } from "@angular/core";

import { HeroeComponent } from './heroe/heroe.component';

import { ListadoComponent } from './listado/listado.component';

@NgModule({

    declarations: [

        HeroeComponent,

        ListadoComponent

    ],

    exports: [

        ListadoComponent

    ],

    imports: [

        CommonModule

    ]

})

export class HeroesModule {}

Los **exports** significan que cosas quiero hacer públicas o visibles fuera de este módulo.

En los **imports** van módulos. Debemos importar el **CommoModule**.

El **CommonModule** se utiliza para poder usar los ng-For y ng-If.

Luego en el app.module.ts debemos borrar los componentes que son de Heroes, porque ya los tenemos dentro de otro módulo, y también debemos importar este nuevo módulo aquí:

import { NgModule } from ‘@angular/core’;

import { BrowserModule } from ‘@angular/platform-browser’;

import { AppComponent } from ‘./app.component’;

import { ContadorComponent } from ‘./contador/contador.component’;

import { HeroesModule } from ‘./éroes/éroes.module’;

@NgModule({

  declarations: [

    AppComponent,

    ContadorComponent,

  ],

  imports: [

    BrowserModule,

    HeroesModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

**SEC-5: EXPANDIENDO NUESTRAS BASES**

**MÓDULO DRAGON BALL Z**

Para crear un módulo desde la terminal corremos el comando:

**ng g m nombreDelModulo**

Luego crear el componente dentro de la carpeta creada por el comando anterior:

**ng g c nombreDelModulo/nombreDelComponente** y si queremos evitar los archivos de Testing le agregamos a este comando **--skipTests**.

**FORMS MODULE**

Debemos importarlo en el módulo en el que lo usemos:

import { FormsModule } from '@angular/forms';

@NgModule({

  declarations: [

    MainPageComponent

  ],

  exports: [

    MainPageComponent

  ],

  imports: [

    CommonModule,

    FormsModule

  ]

})

Y luego aplicarlo en el formulario mediante **ngSubmit**:

<form (ngSubmit)="agregar()">

  <input type="text" placeholder="Nombre" />

   <input type="number" placeholder="Poder" />

   <button type="submit">Agregar</button>

</form>

**NGMODEL**

Para utilizar el ngModel necesitamos tener el atributo name en el input.

<form (ngSubmit)="agregar()">

     <input

        type="text"

        placeholder="Nombre"

        name="name"

        [(ngModel)]="nuevo.name"

     />

            <!-- Otra forma de agarrar el valor del campo -->

            <!-- [value]="nuevo.name"

            (input)="cambiarNombre($event)" -->

     <input

         type="number"

         placeholder="Poder"

         name="power"

         [(ngModel)]="nuevo.power"

     />

**@INPUT**

Se utiliza para pasar información del componente padre al hijo.

**MainPage:**

<app-personajes [personajes]="personajes"></app-personajes>

**Personajes.components.ts:**

export class PersonajesComponent {

  @Input()

  personajes:Personaje[]=[]

}