53 - TypeScript: clases

TypeScript incorpora muchas características de la programación orientada a objetos disponibles en lenguajes como Java y C#.

La sintaxis básica de una clase puede ser:

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

  constructor(nombre:string, edad:number) {
    this.nombre = nombre;
    this.edad = edad;
  }

  imprimir() {
    console.log(`Nombre: ${this.nombre} y edad:${this.edad}`);
  }
}

let persona1: Persona;
persona1 = new Persona('Juan', 45);
persona1.imprimir();
```

Los atributos se definen fuera de los métodos. Para acceder a los mismos dentro de los métodos debemos anteceder la palabra clave 'this':

```
imprimir() {
  console.log(`Nombre: ${this.nombre} y edad:${this.edad}`);
}
```

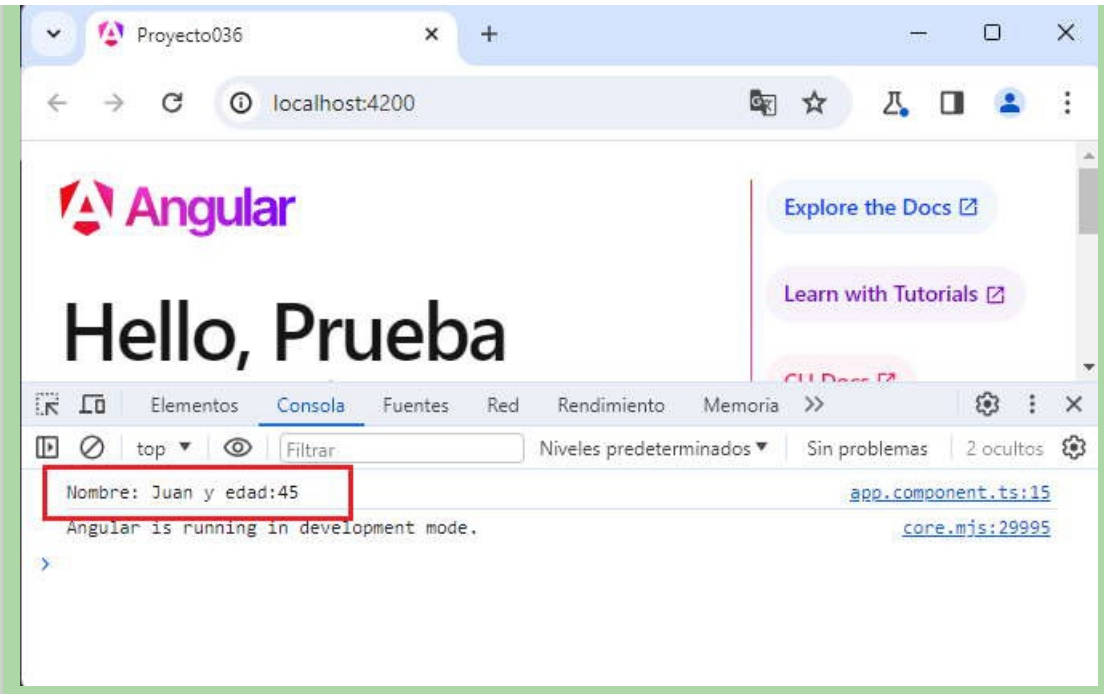
El constructor es el primer método que se ejecuta en forma automática al crear un objeto de la clase 'Persona':

```
constructor(nombre:string, edad:number) {
  this.nombre = nombre;
  this.edad = edad;
}
```

Podemos probar este código agregándolo previo a la componente y luego ver la consola del navegador:

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

class Persona {
  nombre: string;
  edad: number;

  constructor(nombre: string, edad: number) {
    this.nombre = nombre;
    this.edad = edad;
  }

  imprimir() {
    console.log(`Nombre: ${this.nombre} y edad:${this.edad}`);
  }
}

let persona1: Persona;
persona1 = new Persona('Juan', 45);
persona1.imprimir();

@Component({
  selector: 'app-root',
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = "Prueba TypeScript";
}
```

En la consola comprobamos los valores:



Modificadores de acceso a propiedades y métodos.

Podemos definir propiedades y métodos privados y públicos antecediendo las palabras claves 'private' y 'public'.

Veamos un ejemplo:

```
class Dado {
  private valor: number = 1;

  public tirar() {
    this.generar();
  }

  private generar() {
    this.valor = Math.floor(Math.random() * 6) + 1;
  }

  public imprimir() {
    console.log(`Salió el valor ${this.valor}`);
  }
}

let dado1 = new Dado();
dado1.tirar();
dado1.imprimir();
```

El objeto 'dado1' tiene acceso a todos los atributos y métodos que tienen el modificador 'public' por eso podemos llamar a los métodos 'tirar' e 'imprimir':

```
dado1.tirar();
dado1.imprimir();
```

Se generará un error si queremos acceder al atributo 'valor' o al método 'generar':

```
let dado1 = new Dado();
dado1.valor = 6;
dado1.generar();
```

Si no agregamos modificador de acceso por defecto es 'public', luego tendremos el mismo resultado si utilizamos la siguiente sintaxis para declarar la clase:

```
class Dado {
  private valor: number = 1;

  tirar() {
    this.generar();
  }

  private generar() {
    this.valor = Math.floor(Math.random() * 6) + 1;
  }

  imprimir() {
    console.log(`Salió el valor ${this.valor}`);
  }
}

let dado1=new Dado();
dado1.tirar();
dado1.imprimir();
```

Es decir nos ahorramos de escribir 'public' antes de las propiedades y métodos que queremos definir con esta característica.

Todo atributo de una clase debe inicializarse cuando se lo define o en el constructor, sino se genera un error.

Si por algún motivo no queremos inicializar el atributo, debemos utilizar el modificador "!" (Non-null assertion operator)

Podemos indicar a TypeScript que confíe en que la propiedad no será nula utilizando el operador de afirmación de no nulo (!). Sin embargo, debes estar seguro de que nunca intentarás acceder a la propiedad antes de que se le haya asignado un valor:

```
class Dado {
  private valor!: number;

  public tirar() {
    this.generar();
  }

  private generar() {
    this.valor = Math.floor(Math.random() * 6) + 1;
  }

  public imprimir() {
    console.log(`Salió el valor ${this.valor}`);
  }
}

let dado1 = new Dado();
dado1.tirar()
dado1.imprimir()
```

Luego si nos olvidamos de llamar al método 'tirar' y llamamos al método imprimir, se mostrará el valor 'undefined' que es el contenido en el atributo 'valor'.

Definición e inicialización de propiedades en los parámetros del constructor.

Esta característica no es común en otros lenguajes orientados a objetos y tiene por objetivo crear clases más breves.

En TypeScript podemos definir algunas propiedades de la clase en la zona de parámetros del constructor, con esto nos evitamos de su declaración fuera de los métodos.

Por ejemplo la clase Persona la podemos codificar con ésta otra sintaxis:

```
class Persona {

  constructor(public nombre:string, public edad:number) { }

  imprimir() {
    console.log(`Nombre: ${this.nombre} y edad:${this.edad}`);
  }
}

let persona1: Persona;
persona1 = new Persona('Juan', 45);
persona1.imprimir();
```

Como vemos el constructor tiene un bloque de llaves vacías ya que no tenemos que implementar ningún código en su interior, pero al anteceder el modificador de acceso en la zona de parámetros los mismos pasan a ser propiedades de la clase y no parámetros:

```
  constructor(public nombre:string, public edad:number) { }
```

Podemos sin problemas definir propiedades tanto 'public' como 'private'.

La definición de propiedades en la zona de parámetros solo se puede hacer en el constructor de la clase y no está permitido en cualquier otro método.

Modificador readonly

Disponemos además de los modificadores 'private' y 'public' uno llamado 'readonly'. Mediante este modificador el valor de la propiedad solo puede ser cargado en el constructor o al momento de definirlo y luego no puede ser modificado ni desde un método de la clase o fuera de la clase.

Veamos un ejemplo con una propiedad 'readonly':

```
class Articulo {
  readonly codigo: number;
  descripcion: string;
  precio: number;

  constructor(codigo:number, descripcion:string, precio:number) {
    this.codigo=codigo;
    this.descripcion=descripcion;
    this.precio=precio;
  }

  imprimir() {
    console.log(`Código:${this.codigo} Descripción:${this.descripcion} Precio:${this.precio}`);
  }
}

let articulo1: Articulo;
articulo1 = new Articulo(1,'papas',12.5);
articulo1.imprimir();
```

Una vez que se inicia la propiedad 'codigo' en el constructor su valor no puede cambiar:

```
  imprimir() {
    this.codigo=7; //Error
    console.log(`Código:${this.codigo} Descripción:${this.descripcion} Precio:${this.precio}`);
  }
```

El mismo error se produce si tratamos de cambiar su valor desde fuera de la clase:

```
let articulo1: Articulo;
articulo1 = new Articulo(1,'papas',12.5);
articulo1.codigo=7; //Error
articulo1.imprimir();
```

Podemos utilizar también la sintaxis abreviada de propiedades:

```
class Articulo {

  constructor(readonly codigo:number, public descripcion:string, public precio:number) { }

  imprimir() {
    console.log(`Código:${this.codigo} Descripción:${this.descripcion} Precio:${this.precio}`);
  }
}
```

Propiedades estáticas

Las propiedades estáticas pertenecen a la clase y no a las instancias de la clase. Se las define antecediendo el modificador 'static'.

Con un ejemplo quedará claro este tipo de propiedades:

```
class Dado {
  private valor: number = 1;
  static tiradas:number=0;
  tirar() {
    this.generar();
  }

  private generar() {
    this.valor = Math.floor(Math.random() * 6) + 1 ;
    Dado.tiradas++;
  }

  imprimir() {
    console.log(`Salió el valor ${this.valor}`);
  }
}

let dado1=new Dado();
dado1.tirar();
dado1.imprimir();
let dado2=new Dado();
dado2.tirar();
dado2.imprimir();
console.log(`Cantidad de tiradas de dados:${Dado.tiradas}`); // 2
```

Una propiedad estática requiere el modificador 'static' previo a su nombre:

```
  static tiradas:number=0;
```

Para acceder a dichas propiedades debemos anteceder el nombre de la clase y no la palabra clave 'this':

```
    Dado.tiradas++;
```

No importan cuantos objetos de la clase se definan luego todos esos objetos comparten la misma variable estática:

```
let dado1=new Dado();
dado1.tirar();
dado1.imprimir();
let dado2=new Dado();
dado2.tirar();
dado2.imprimir();
console.log(`Cantidad de tiradas de dados:${Dado.tiradas}`); // 2
```

Es por eso que la propiedad 'tiradas' almacena un 2 luego de tirar el primer y segundo dado.

La propiedad 'valor' es independiente en cada dado pero la propiedad 'tiradas' es compartida por los dos objetos.

Métodos estáticas

Igual que las propiedades estáticas los métodos estáticos se los accede por el nombre de la clase. Este tipo de métodos solo pueden acceder a propiedades estáticas.

```
class Matematica {
  static mayor(v1:number, v2: number): number {
    if (v1>v2)
      return v1;
    else
      return v2;
  }

  static menor(v1:number, v2: number): number {
    if (v1<v2)
      return v1;
    else
      return v2;
  }

  static aleatorio(inicio: number, fin: number): number {
    return Math.floor((Math.random()*(fin+1-inicio))+inicio);
  }
}

let x1=Matematica.aleatorio(1,10);
let x2=Matematica.aleatorio(1,10);
console.log(`El mayor entre ${x1} y ${x2} es ${Matematica.mayor(x1,x2)}
`);
console.log(`El menor entre ${x1} y ${x2} es ${Matematica.menor(x1,x2)}`);
```

Debemos anteceder la palabra clave static al nombre del método.

Cuando llamamos a un método debemos anteceder también el nombre de la clase, no hace falta definir una instancia u objeto de la clase:

```
let x1=Matematica.aleatorio(1,10);
```

Retornar