POO en TS

Antonio Espín Herranz

Contenidos

- Clases
- Propiedades
- Métodos
- Constructores
- Modificadores de acceso
- Encapsulación
- Herencia
- Interfaces
- Clases Abstractas, polimorfismo
- Métodos estáticos

Clases

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

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

  saludar(): void {
    console.log(`Hola, mi nombre es ${this.nombre} y tengo ${this.edad} años.`);
  }
}

// Crear una instancia de la clase
const persona1 = new Persona("Antonio", 30);
persona1.saludar();
// Salida: Hola, mi nombre es Antonio y tengo 30 años.
```

Constructores

- En TypeScript, no se permite la sobrecarga de constructores de manera directa como ocurre en otros lenguajes orientados a objetos como Java. Sin embargo, se puede simular la sobrecarga mediante un único constructor y el uso de tipos opcionales o uniones de tipos para manejar diferentes casos.

Ejemplo

```
class Persona {
 nombre: string;
 edad?: number; // Propiedad opcional

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

```
 mostrarInfo(): string {
   return this.edad !== undefined
    ? `${this.nombre}, ${this.edad} años`
    : `${this.nombre}`;
  }
}
```

- // Crear instancias
- const persona1 = new Persona("Antonio");
- const persona2 = new Persona("María", 28);

- console.log(persona1.mostrarInfo()); // "Antonio"
- console.log(persona2.mostrarInfo()); // "María, 28 años"

Constructores

- Firmas sobrecargadas (constructor(...)):
- Especificas las firmas posibles para el constructor (tipos de parámetros y cantidad de argumentos).
- Estas firmas son declarativas y no implementan lógica.
- Único constructor implementado:
- Solo puedes implementar un constructor con la lógica completa.
- Dentro del constructor, manejas las posibles combinaciones de parámetros con validaciones (if, undefined, etc.).
- También se pueden utilizar parámetros opcionales

Encapsulación

- La encapsulación en TypeScript es un principio fundamental de la Programación Orientada a Objetos (POO) que busca restringir el acceso directo a los datos y exponer únicamente las operaciones necesarias mediante métodos o propiedades controladas.
- Esto mejora la seguridad, la modularidad y el mantenimiento del código.

Get / Set

- En TypeScript, la encapsulación mediante getters y setters permite controlar cómo se accede y modifica el estado interno de una clase. Esto proporciona un nivel de control adicional al validar o realizar lógica cuando se recuperan o establecen valores en propiedades privadas

Ejemplo

```
class Persona {
 private _nombre: string; // Propiedad privada con convención "_"
 private _edad: number;
 constructor(nombre: string, edad: number) {
  this._nombre = nombre;
  this._edad = edad;
 }
get nombre(): string {
  return this._nombre;
 }
 set nombre(nuevoNombre: string) {
  if (nuevoNombre.trim() === "") {
   throw new Error("El nombre no puede estar vacío.");
  }
  this._nombre = nuevoNombre;
 }
get edad(): number {
  return this._edad;
 }
set edad(nuevaEdad: number) {
  if (nuevaEdad < 0) {
   throw new Error("La edad no puede ser negativa.");
  }
  this._edad = nuevaEdad;
 }
}
```

```
// Crear una instancia
const persona = new Persona("Antonio", 30);
// Usar getters
console.log(persona.nombre); // "Antonio"
console.log(persona.edad);   // 30
// Usar setters
persona.nombre = "María";
persona.edad = 28;
console.log(persona.nombre); // "María"
console.log(persona.edad);   // 28
// Validación en setters
try {
 persona.nombre = ""; // Lanza un error: "El nombre no puede estar vacío."
} catch (error) {
 console.log(error.message);
}
try {
 persona.edad = -5; // Lanza un error: "La edad no puede ser negativa."
} catch (error) {
 console.log(error.message);
}
```

Herencia

TypeScript solo soporta herencia simple:

```
class Empleado extends Persona {
  puesto: string;

  constructor(nombre: string, edad: number, puesto: string) {
    super(nombre, edad); // Llama al constructor de la clase padre
    this.puesto = puesto;
  }

  trabajar(): void {
    console.log(` ${this.nombre} está trabajando como ${this.puesto}. `);
  }
}

// Crear una instancia
const empleado = new Empleado("María", 28, "Desarrolladora");
empleado.saludar();
empleado.trabajar();
```

Interfaces

```
interface Animal {
  nombre: string;
  emitirSonido(): void;
}

class Perro implements Animal {
  nombre: string;

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

  emitirSonido(): void {
    console.log("¡Guau guau!");
  }
}

const miPerro = new Perro("Rex");
miPerro.emitirSonido();
```

Interfaces

- **Una clase puede implementar varios interfaces:**

```
interface Volador {
  volar(): void;
}

interface Nadador {
  nadar(): void;
}

class Pato implements Volador, Nadador {
  volar(): void {
    console.log("El pato está volando.");
  }

  nadar(): void {
    console.log("El pato está nadando.");
  }
}

const pato = new Pato();
pato.volar(); // "El pato está volando."
pato.nadar(); // "El pato está nadando."
```

Modificadores de acceso

- TypeScript proporciona public, private y protected para controlar el acceso a las propiedades y métodos:

- **public**: Accesible desde cualquier lugar (por defecto).

- **private**: Solo accesible dentro de la clase.

- **protected**: Accesible dentro de la clase y sus subclases.

Modificadores de acceso

```
class CuentaBancaria {
 private saldo: number;

 constructor(saldoInicial: number) {
  this.saldo = saldoInicial;
 }

 depositar(cantidad: number): void {
  this.saldo += cantidad;
 }

 consultarSaldo(): number {
  return this.saldo;
 }
}

const cuenta = new CuentaBancaria(1000);
cuenta.depositar(500);
console.log(cuenta.consultarSaldo()); // Salida: 1500
```

Modificadores de acceso

- Se pueden definir métodos y propiedades privadas utilizando # delante del nombre.

```
class CuentaBancaria {
  #saldo: number; // Propiedad privada
  #titular: string; // Propiedad privada

  constructor(titular: string, saldoInicial: number) {
    this.#titular = titular;
    this.#saldo = saldoInicial;
  }

  // Método público para consultar el saldo
  consultarSaldo(): string {
    return `El saldo de ${this.#titular} es ${this.#saldo}€`;
  }

  // Método público para depositar dinero
  depositar(cantidad: number): void {
    if (cantidad <= 0) {
      console.log("La cantidad debe ser mayor que cero.");
      return;
    }
    this.#saldo += cantidad;
    console.log(`Has depositado ${cantidad}€. Nuevo saldo: ${this.#saldo}€.`);
  }

  // Método privado para validar operaciones (ejemplo ficticio)
  #validarOperacion(cantidad: number): boolean {
    return cantidad > 0;
  }
}
```

Modificadores de acceso

- Código principal

- // Crear una instancia de la clase
- const cuenta = new CuentaBancaria("Antonio", 1000);

- // Acceder a métodos públicos
- cuenta.depositar(500);
- console.log(cuenta.consultarSaldo());

- // Intentar acceder a propiedades privadas (esto da error)
- console.log(cuenta.#saldo); **// Error: Property '#saldo' is not accessible**

Clases Abstractas, polimorfismo

- Polimorfismo: la capacidad que tienen los objetos de responder al mismo método reflejando el comportamiento distinto dependiendo de la clase a la que pertenecen. Está relacionado con la herencia.

Ejemplo

```
abstract class Figura {
  abstract calcularArea(): number;
}

class Circulo extends Figura {
  radio: number;

  constructor(radio: number) {
    super();
    this.radio = radio;
  }

  calcularArea(): number {
    return Math.PI * this.radio * this.radio;
  }
}
```

```
class Rectangulo extends Figura {
  ancho: number;
  alto: number;

  constructor(ancho: number, alto: number) {
    super();
    this.ancho = ancho;
    this.alto = alto;
  }

  calcularArea(): number {
    return this.ancho * this.alto;
  }
}

const figuras: Figura[] = [new Circulo(5), new Rectangulo(4, 7)];
figuras.forEach((figura) => console.log(figura.calcularArea()));
```

Métodos static

- **Características de los métodos estáticos:**
  - **Acceso sin instanciación**: Los métodos estáticos se llaman directamente usando el nombre de la clase.
  - No pueden acceder a propiedades o métodos de instancia:
  - Los métodos estáticos solo pueden acceder a otros métodos estáticos o propiedades estáticas dentro de la misma clase.
  - Intentar acceder a propiedades o métodos de instancia dentro de un método estático dará error.

Métodos static

- **Ventajas de los métodos estáticos:**

- Útiles para funcionalidades que no dependen de los datos o estado de una instancia.

- Ideal para utilidades, helpers o funciones matemáticas, como los ejemplos mostrados

Métodos static

```
class Calculadora {
  // Método estático para sumar números
  static sumar(a: number, b: number): number {
    return a + b;
  }

  // Método estático para multiplicar números
  static multiplicar(a: number, b: number): number {
    return a * b;
  }
}

// Llamada al método estático directamente desde la clase
console.log(Calculadora.sumar(5, 3)); // Salida: 8
console.log(Calculadora.multiplicar(4, 7)); // Salida: 28
```

Propiedades static

```
class Utilidades {
  static version: string = "1.0";

  static mostrarVersion(): void {
    console.log(`Versión actual: ${this.version}`);
  }
}
```

- // Llamada al método estático y acceso a propiedad estática
- Utilidades.mostrarVersion(); // Salida: Versión actual: 1.0