TypeScript

Programación Orientada a Objetos (POO)

CertiDevs

Índice de contenidos

1. Programación Orientada a Objetos 1
2. Clases 2
    2.1. Modificadores de acceso 2
3. Herencia 3
4. Interfaces 4
    4.1. Definición de interfaces 4
    4.2. Implementación de interfaces 4
    4.3. Extensión de interfaces 5
    4.4. Interfaces para definición de tipos de objetos 6
5. Composición 6
    5.1. Asociaciones de uno a uno 6
    5.2. Relaciones uno a muchos 8
    5.3. Relaciones muchos a muchos 9
6. Abstracción 10
7. Polimorfismo 11
8. Importar y exportar módulos 12

1. Programación Orientada a Objetos

TypeScript es un lenguaje de programación basado en **objetos** que **extiende** el sistema de objetos de JavaScript con características adicionales como *tipado estático*, *clases*, *interfaces*, *modificadores de acceso*, *herencia* y más.

Programación Orientada a Objetos (OOP) es un paradigma de programación que utiliza objetos y clases para estructurar y organizar el código.

La **OOP** es un paradigma de programación que se basa en la organización y estructuración del código utilizando **objetos**, que son entidades que encapsulan **datos** y **comportamientos** relacionados. Este enfoque permite diseñar soluciones de software más **modulares**, **reutilizables** y **escalables**.

La POO se basa en varios conceptos clave, que incluyen:

- **Clases**: Las clases son plantillas o "moldes" que definen la estructura y el comportamiento de los objetos. Una clase especifica los atributos (datos) y métodos (funciones) que los objetos creados a partir de esa clase tendrán.
- **Objetos**: Los objetos son instancias de clases. Cada objeto creado a partir de una clase tiene su propio conjunto de atributos y puede utilizar los métodos definidos en la clase.
- **Encapsulamiento**: El encapsulamiento es un principio que permite ocultar los detalles de implementación de una clase y controlar el acceso a sus atributos y métodos. Esto facilita el mantenimiento y mejora la modularidad del código.
- **Herencia**: La herencia es un mecanismo que permite crear una nueva clase basada en otra existente. La nueva clase hereda los atributos y métodos de la clase base y puede agregar o sobrescribir características según sea necesario.
- **Polimorfismo**: El polimorfismo permite que diferentes clases tengan métodos con el mismo nombre pero con comportamientos diferentes. Esto facilita la reutilización de código y permite diseñar interfaces más flexibles.
- **Composición**: La composición se logra al incluir objetos de otras clases como atributos dentro de una clase. Esto permite que una clase tenga acceso a los atributos y métodos de otras clases, lo que facilita la reutilización y el mantenimiento del código.

Aprender POO tiene varias **ventajas**:

- **Modularidad**: La POO facilita la organización del código en componentes independientes y fácilmente intercambiables. Esto facilita el mantenimiento y la extensión del código a lo largo del tiempo.
- **Reutilización de código**: La POO permite reutilizar y compartir código a través de la herencia y la composición de clases. Esto reduce la duplicación de código y mejora la eficiencia del desarrollo.
- **Abstracción**: La POO permite abstraer y modelar conceptos del mundo real en términos de objetos y clases. Esto facilita la comprensión y el diseño de soluciones de software.
- **Flexibilidad**: La POO permite diseñar sistemas de software que son más fáciles de adaptar y

extender a medida que cambian los requisitos. Esto es especialmente útil en proyectos de desarrollo de software a largo plazo o en equipos de desarrollo grandes.

2. Clases

Las **clases** en TypeScript son la base de la programación orientada a objetos y permiten definir plantillas para crear objetos con propiedades y métodos específicos.

```
class Person {
    firstName: string;
    age: number;

    constructor(firstName: string, age: number) {
        this.firstName = firstName;
        this.age = age;
    }

    sayHello(): void {
        console.log(`Hello, my name is ${this.firstName}.`);
    }
}

const alice = new Person('Alice', 30);
alice.sayHello(); // Hello, my name is Alice.
```

2.1. Modificadores de acceso

TypeScript admite **modificadores de acceso** para controlar la **visibilidad** y el acceso a **propiedades** y **métodos** de las clases.

Los modificadores de acceso disponibles en TypeScript son `public`, `private` y `protected`.

- `public`: Las propiedades y métodos son accesibles desde cualquier parte del código. Este es el modificador de acceso predeterminado en TypeScript.
- `private`: Las propiedades y métodos solo son accesibles dentro de la clase que los define.
- `protected`: Las propiedades y métodos son accesibles dentro de la clase que los define y en sus subclases.

```
class Person {

    constructor(public firstName: string, public age: number) {}

    public sayHello(): void {
        console.log(`Hello, my name is ${this.firstName}.`);
    }
```

```
}
```

No es necesario agregar `this.name = name;` y `this.age = age;` en el **constructor** de la clase `Person`, al utilizar el modificador `public` en los parámetros del **constructor** en **TypeScript** automáticamente se declaran y asignan las propiedades correspondientes al objeto que se está creando.

En cambio, en **JavaScript**, no hay concepto de modificadores de acceso (`public`, `private`, `protected`) como en **TypeScript**. Por defecto, todos los atributos son accesibles desde cualquier parte del código si se utilizan con una instancia del objeto. Para definir atributos en una clase de JavaScript, es necesario utilizar `this` en el constructor, ya que no hay sintaxis de modificadores de acceso en los parámetros del constructor como en TypeScript.

Por otro lado, en **TypeScript**, si los atributos son privados (`private`), deberás agregar el modificador `private` en lugar de `public` en los parámetros del constructor.

```
class Person {

    constructor(private firstName: string, private age: number) {}

    sayHello(): void {
        console.log(`Hello, my name is ${this.firstName} with age: ${this.age}`);
    }
}

const alice = new Person('Alice', 30);
alice.sayHello(); // Hello, my name is Alice.
```

3. Herencia

La **herencia** en TypeScript permite crear **clases** que **extienden** otras **clases**, lo que permite reutilizar y extender el código.

La herencia se logra utilizando la palabra clave `extends`.

```
class Person {

    constructor(public firstName: string, public age: number) {}

    public sayHello(): void {
        console.log(`Hello, my name is ${this.firstName}.`);
    }

}

class Employee extends Person {

    constructor(public firstName: string, public age: number, public position: string)
```

```
{
        super(firstName, age);
        this.position = position;
    }

    sayHello(): void {
        super.sayHello();
        console.log(`I work as a ${this.position}.`);
    }
}

const bob = new Employee('Bob', 35, 'developer');
bob.sayHello(); // Hello, my name is Bob. I work as a developer.
```

4. Interfaces

Las **interfaces** sirven para definir contratos que deben cumplir las clases y objetos en el código.

Las **interfaces** pueden especificar propiedades y métodos que deben ser implementados por las clases que implementan la interfaz. Además, las interfaces también pueden ser utilizadas para la definición de tipos de objetos.

4.1. Definición de interfaces

Una **interfaz** se define utilizando la palabra clave `interface` seguida del nombre de la interfaz y un conjunto de propiedades y métodos.

Las **propiedades** se definen con su nombre y tipo, mientras que los **métodos** se definen con su nombre, parámetros y tipo de retorno.

```
interface IPerson {
    name: string;
    age: number;
    sayHello(): void;
}
```

4.2. Implementación de interfaces

Las **clases** pueden **implementar interfaces** utilizando la palabra clave `implements`.

Cuando una **clase** implementa una **interfaz**, debe proporcionar una **implementación** para todas las **propiedades** y **métodos** definidos en la interfaz.

```
interface IPerson {
    name: string;
    age: number;
```

```
    sayHello(): void;
}

class Person implements IPerson {

    // No es necesario agregar this.name = name; y this.age = age; en el constructor
    constructor(public name: string, public age: number) {}

    sayHello(): void {
        console.log(`Hello, my name is ${this.name}.`);
    }
}

const alice = new Person('Alice', 30);
alice.sayHello(); // Hello, my name is Alice.
```

4.3. Extensión de interfaces

Las **interfaces** pueden extender otras interfaces utilizando la palabra clave `extends`.

Esto permite crear interfaces más específicas a partir de interfaces más generales, lo que facilita la reutilización y composición de interfaces.

```
interface IPerson {
    name: string;
    age: number;
    sayHello(): void;
}

interface IEmployee extends IPerson {
    position: string;
    work(): void;
}

class Employee implements IEmployee {

    constructor(public name: string,
                public age: number,
                public position: string) {}

    sayHello(): void {
        console.log(`Hello, my name is ${this.name}.`);
    }

    work(): void {
        console.log(`I work as a ${this.position}.`);
    }
}
```

```
const bob = new Employee('Bob', 35, 'developer');
bob.work(); // I work as a developer.
```

4.4. Interfaces para definición de tipos de objetos

Las **interfaces** también pueden ser utilizadas para definir tipos de objetos sin necesidad de implementar una clase.

Esto es útil para describir la estructura de objetos que no requieren una implementación específica de una clase.

```
interface ICar {
    brand: string;
    model: string;
    year: number;
}

const car: ICar = {
    brand: 'Toyota',
    model: 'Corolla',
    year: 2019,
};

console.log(car.brand); // Toyota
```

En este ejemplo, la interfaz ICar define un tipo de objeto con las propiedades `brand`, `model` y `year`.

Se puede crear un objeto que cumpla con este tipo sin necesidad de implementar una clase específica.

5. Composición

Las **interfaces** en TypeScript también pueden utilizarse para definir la estructura de objetos en la **capa de modelo** de una aplicación.

Esto permite describir las **relaciones** entre los diferentes tipos de objetos sin necesidad de implementar una clase específica para cada uno de ellos.

A continuación, se muestran ejemplos de cómo utilizar interfaces para modelar relaciones/asociaciones **uno a uno**, **uno a muchos** y **muchos a muchos** en TypeScript, de forma que logremos la **composición**, donde un objeto puede estar asociado a otro objeto para crear estructuras más complejas.

5.1. Asociaciones de uno a uno

En este ejemplo, se definen interfaces para modelar una **relación uno a uno** (one to one) entre una clase `User` y una clase `Profile`

```
// asociación one to one unidireccional

interface IUser {
    id: number;
    name: string;
    profile: IProfile;
}

interface IProfile {
    id: number;
    bio: string;
}


const user: IUser = {
    id: 1,
    name: 'Alice',
    profile: {
        id: 1,
        bio: 'I love programming!'
    },
};

console.log(user.profile.bio); // I love programming!
```

```
// asociación one to one bidireccional

interface IUser {
    id: number;
    name: string;
    profile: IProfile;
}

interface IProfile {
    id: number;
    bio: string;
    user: IUser;
}


const user: IUser = {
    id: 1,
    name: 'Alice',
    profile: {
        id: 1,
        bio: 'I love programming!',
        user: null, // Esta propiedad se rellenará más adelante
    },
};
```

```
user.profile.user = user; // Establecemos la relación bidireccional entre User y
Profile

console.log(user.profile.bio); // I love programming!
console.log(user.profile.user.name); // Alice


// opción 2: crear una clase que implemente la interfaz IProfile
class Profile implements IProfile {
    constructor(public id: number, public bio: string, public user: IUser) {}
}

const profile = new Profile(1, 'I love programming!', user);
user.profile = profile;

console.log(user.profile.bio); // I love programming!
```

En este caso, las interfaces `IUser` e `IProfile` describen la relación uno a uno entre un usuario y su perfil.

La propiedad `profile` en `IUser` es de tipo `IProfile`, mientras que la propiedad user en `IProfile` es de tipo `IUser`.

5.2. Relaciones uno a muchos

Ahora se definen interfaces para modelar `una relación uno a muchos` entre una clase `Author` y una clase `Book`.

```
interface IAuthor {
    id: number;
    name: string;
    books: IBook[];
}

interface IBook {
    id: number;
    title: string;
    author: IAuthor;
}

const author: IAuthor = {
    id: 1,
    name: 'George Orwell',
    books: [], // Esta propiedad se rellenará más adelante
};

const book1: IBook = {
    id: 1,
```

```
    title: '1984',
    author: author,
};

const book2: IBook = {
    id: 2,
    title: 'Animal Farm',
    author: author,
};

author.books.push(book1, book2); // Establecemos la relación entre Author y sus Books

console.log(author.books[0].title); // 1984
console.log(book1.author.name); // George Orwell
```

En este ejemplo, la interfaz `IAuthor` tiene una propiedad `books`, que es un arreglo de objetos `IBook`.

La interfaz `IBook` tiene una propiedad author de tipo `IAuthor`. Esto permite modelar una relación uno a muchos entre un autor y sus libros.

5.3. Relaciones muchos a muchos

Por último, se definen interfaces para modelar una **relación muchos a muchos** entre una clase `Student` y una clase `Course`.

```
interface IStudent {
    id: number;
    name: string;
    courses: ICourse[];
}

interface ICourse {
    id: number;
    title: string;
    students: IStudent[];
}

const student1: IStudent = {
    id: 1,
    name: 'Alice',
    courses: [], // Esta propiedad se rellenará más adelante
};

const student2: IStudent = {
    id: 2,
    name: 'Bob',
    courses: [], // Esta propiedad se rellenará más adelante
};
```

```
const course1: ICourse = {
    id: 1,
    title: 'Mathematics',
    students: [], // Esta propiedad se rellenará más adelante
};

const course2: ICourse = {
    id: 2,
    title: 'Physics',
    students: [], // Esta propiedad se rellenará más adelante
};

// Establecemos las relaciones entre estudiantes y cursos
student1.courses.push(course1, course2);
student2.courses.push(course1);

course1.students.push(student1, student2);
course2.students.push(student1);

console.log(student1.courses[1].title); // Physics
console.log(course1.students[1].name); // Bob
```

En este ejemplo, las interfaces `IStudent` e `ICourse` describen una relación muchos a muchos entre estudiantes y cursos.

La propiedad `courses` en `IStudent` es un arreglo de objetos `ICourse`, mientras que la propiedad `students` en `ICourse` es un arreglo de objetos `IStudent`. Esto permite modelar una relación muchos a muchos entre estudiantes y cursos.

6. Abstracción

TypeScript admite la creación de clases y métodos **abstractos** mediante la palabra clave `abstract`.

Las **clases abstractas** no pueden ser instanciadas directamente y deben ser extendidas por otras clases.

Los **métodos abstractos** no tienen implementación y deben ser implementados por las clases derivadas.

```
abstract class Animal {
    abstract makeSound(): void;
}

class Dog extends Animal {
    makeSound(): void {
        console.log('Woof! Woof!');
    }
}
```

```
class Cat extends Animal {
    makeSound(): void {
        console.log('Meow! Meow!');
    }
}

const dog = new Dog();
dog.makeSound(); // Woof! Woof!

const cat = new Cat();
cat.makeSound(); // Meow! Meow!
```

7. Polimorfismo

El **polimorfismo** en TypeScript permite tratar objetos de diferentes clases como si fueran objetos de una clase común.

Esto es especialmente útil cuando se trabaja con clases que implementan una misma interfaz o que extienden una misma clase base.

```
interface IShape {
    area(): number;
}

class Circle implements IShape {
    constructor(public radius: number) {}

    area(): number {
        return Math.PI * this.radius ** 2;
    }
}

class Rectangle implements IShape {
    constructor(public width: number, public height: number) {}

    area(): number {
        return this.width * this.height;
    }
}

const shapes: IShape[] = [
    new Circle(5),
    new Rectangle(10, 20),
];

shapes.forEach(shape => console.log(shape.area()));
```

En este ejemplo, tanto Circle como Rectangle implementan la interfaz IShape. Se puede utilizar el

polimorfismo para tratar objetos de ambas clases como objetos de tipo IShape y llamar al método area() en ambos casos.

8. Importar y exportar módulos

En TypeScript, puedes utilizar clases definidas en otros archivos utilizando **módulos** y **exportaciones** e **importaciones**.

Crea una clase en un archivo TypeScript y utiliza modificadores de visibilidad para sus miembros.

Por ejemplo, crea un archivo llamado `person.ts` y define una clase `Person` con propiedades y métodos con diferentes modificadores de visibilidad:

```
// person.ts

export class Person {
    public firstName: string;
    private lastName: string;
    protected age: number;

    constructor(firstName: string, lastName: string, age: number) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    public greet(): void {
        console.log(`Hello, my name is ${this.firstName} ${this.lastName}.`);
    }

    private getFullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }

    protected getAge(): number {
        return this.age;
    }
}
```

En este ejemplo, utilizamos los modificadores de visibilidad `public`, `private` y `protected` para diferentes miembros de la clase `Person`.

También utilizamos la palabra clave `export` antes de la declaración de la clase para que pueda ser importada y utilizada en otros archivos.

Importa y utiliza la clase en otro archivo TypeScript.

En otro archivo TypeScript, importa y utiliza la clase Person definida en person.ts.

Por ejemplo, crea un archivo llamado main.ts:

```
// main.ts

import { Person } from './person';

const person = new Person('John', 'Doe', 30);
person.greet(); // Hello, my name is John Doe.

// Error: Property 'lastName' is private and only accessible within class 'Person'.
// console.log(person.lastName);

// Error: Property 'getFullName' is private and only accessible within class 'Person'.
// console.log(person.getFullName());

// Error: Property 'age' is protected and only accessible within class 'Person' and
its subclasses.
// console.log(person.age);
// Error: Property 'getAge' is protected and only accessible within class 'Person' and
its subclasses.
// console.log(person.getAge());
```

En este ejemplo, importamos la clase `Person` desde el archivo person.ts utilizando la declaración de importación.

Luego, creamos una instancia de la clase `Person` y llamamos al método `greet()`. No podemos acceder a los miembros privados y protegidos de la clase Person desde fuera de la clase o sus subclases.

Estos pasos te permiten utilizar clases TypeScript definidas en otros archivos, respetando los modificadores de visibilidad para garantizar el encapsulamiento y el acceso adecuado a los miembros de la clase.