

Prog. Orientada a Objetos

Carrera Programador full-stack

Interfaces + abstract

Agenda

- Importancia del planteo de la solución
- Concepto de interfaces
- Ejemplo de la interfaz Auto
- Restricciones de las Interfaces
- Concepto de clase abstracta
- Recomendaciones
- Ejercicios

Importancia de plantear la Solución

- Una de las ideas centrales de POO es querer que una clase haga algo sin que me importe la forma en que lo hace
- Recordar la forma original de plantear una clase
 - Variables internas
 - Métodos
- Es posible hacer un planteo de esta manera en TypeScript sin definir la implementación
- De esta manera podemos crear una clase teniendo *mucho más presente* el planteo de lo que debería hacer

Concepto de Interfaces

- En TypeScript se pueden plantear los métodos que una clase *debe* tener
- Estos “planteos” se llaman *interfaces*
- Una clase puede *heredar* de otra clase
- Una clase **también** puede *implementar* una interfaz
- Las interfaces pueden verse como “contratos” que las clases deben cumplir
 - En caso de no cumplirlo → tsc no compila
 - Los métodos que define la interfaz, deben ser implementados en la clase

Qué es una interfaz?


Una interfaz es un contrato entre dos entidades, esto quiere decir que una interfaz provee un servicio a una clase consumidora.

Por ende, la interfaz solo nos muestra la declaración de los métodos que esta posee, no su implementación, permitiendo así su encapsulamiento.

Ejemplo de la Interfaz Auto (1)

```
interface Auto {  
    acelerar(): void;  
    getVelocidadActual(): number;  
}
```

**Solamente definimos los
métodos, no los implementamos**



Tampoco se pueden definir los modificadores de acceso



En caso de ponerlos → tsc se queja



Ejemplo de la Interfaz Auto (2)

```
interface Auto {  
    acelerar(): void;  
    getVelocidadActual(): number;  
}
```

**La clase AutoCarreras
implementa los métodos de la
interfaz Auto**

```
class AutoCarreras implements Auto {  
    private velocidadActual: number;  
  
    public constructor() {  
        this.velocidadActual = 0;  
    }  
  
    public acelerar(): void {  
        this.velocidadActual += 50;  
    }  
  
    public getVelocidadActual(): number {  
        return this.velocidadActual;  
    }  
}
```

**Se pueden agregar más variables/métodos siempre y cuando
se respete el “contrato”**

Restricciones de las Interfaces

- No permite implementar los métodos → definición de interfaz
- No permite modificadores de acceso, todo lo que se plantee es *public*
 - Esto responde a la idea de conocer desde afuera **qué** es lo que hacen las clases → no tiene sentido que se puedan definir cosas privadas
- Si bien las interfaces permiten definir variables, no es recomendable porque quedan públicas
 - Se rompe la abstracción!

Interfaces - Demostración de Vivo

- Definir la interfaz Auto
- Implementar la clase AutoCarrera
- Implementar la clase AutoFamiliar
- Forzar algunos errores típicos



Clases Abstractas

- Existe un “punto intermedio” entre las clases y las interfaces
 - Ni todo definido (interfaces), ni todo implementado (clases)
- Las *clases abstractas* son clases normales, pero que permiten tener métodos *sin implementar*
- La contra que tienen, es que no se pueden instanciar → o sea no se pueden usar
 - Tiene que haber otra clase que las extienda e implemente el método que les falta implementar
- Una clase no se puede usar si no tiene *todos* sus métodos implementados

Clases Abstractas - Ejemplo (1)

```
abstract class AutoCiudad {  
    protected velocidadActual: number;  
    private estaPrendido: boolean;  
  
    public constructor() {  
        this.velocidadActual = 0;  
        this.estaPrendido = false;  
    }  
  
    abstract acelerar(): void;  
  
    public prender(): void {  
        this.estaPrendido = true;  
    }  
  
    public apagar(): void {  
        this.estaPrendido = false;  
    }  
}
```

```
class AutoCiudadChico extends AutoCiudad {  
    public acelerar(): void {  
        this.velocidadActual += 10;  
    }  
}
```

Una clase abstracta tiene *mínimo* un método abstracto

```
    public prender(): void {  
        this.estaPrendido = true;  
    }  
  
    public apagar(): void {  
        this.estaPrendido = false;  
    }  
}
```

Clases Abstractas - Ejemplo (2)

```
abstract class AutoCiudad {  
    protected velocidadActual: number;  
    private estaPrendido: boolean;  
  
    public constructor() {  
        this.velocidadActual = 0;  
        this.estaPrendido = false;  
    }  
  
    abstract acelerar(): void;  
    ...  
}
```

```
let autoCiudad: AutoCiudad = new AutoCiudad();
```

```
PS C:\Users\Francisco\Documents\CFP\3. POO\Ejercicios\poo\interfaces> tsc auto.ts; node auto.js  
auto.ts:42:30 - error TS2511: Cannot create an instance of an abstract class.
```

```
42 let autoCiudad: AutoCiudad = new AutoCiudad();  
                                ~~~~~
```

Found 1 error.

Clases Abstractas - Demo en Vivo

- Implementación del ejemplo anterior
- Forzar algún error

Interfaz vs. Clase Abstracta

- Se usa *interfaz* cuando queremos mostrar *solamente* la funcionalidad de una determinada clase, o un conjunto de clases
- Las *clases abstractas* se usan cuando queremos abstraer más cosas → variables, métodos privados, etc.
- El uso de interfaz es una buena práctica para plantear y mostrar fácilmente la funcionalidad que ofrecen las clases que implementen dicha interfaz → es muy recomendable usarlas

Recomendaciones

- Muy importante el uso de la interfaz → ayuda a plantear lo que queremos hacer
- El uso de clase abstracta es útil cuando tenemos clases con las mismas variables, y queremos abstraerlas
 - Y los métodos que *no podemos abstraer*, ponerlos como abstractos en la *superclase*
- Tener paciencia con los conceptos, no son fáciles
 - Se van madurando de a poco
- Tener cuidado con las interfaces, TypeScript permite definir variables, pero todo es público en una interfaz
 - Por lo tanto cuando una clase la implemente, va a tener una variable interna pública → Evitarlo!

Prog. Orientada a Objetos

Carrera Programador full-stack

Ejercicios

Ejercicios – en clase

Ejercicio 2

- Implementar las siguientes clases con las variables y métodos que crea necesarios:
 - AutoCiudad
 - AutoDeportivo
 - Camioneta
- Abstraer elementos en común entre dichas clases → pasarlos a una clase abstracta, y que las tres clases extiendan de ella

Ejercicios en clase

- Crea una **clase abstracta** llamada DispositivoElectronico con las siguientes propiedades y métodos:
- Propiedades:
 - nombre: string
 - estado: boolean (indica si está encendido o apagado)
- Métodos:
 - encender(): Cambia el estado a true y muestra un mensaje indicando que el dispositivo está encendido.
 - apagar(): Cambia el estado a false y muestra un mensaje indicando que el dispositivo está apagado.
 - funcionalidadEspecial(): Método abstracto que será implementado por las subclases.
- Crea dos clases que hereden de DispositivoElectronico:
- Television: En su funcionalidad especial, debe mostrar un mensaje que diga "Cambiando de canal".
- Radio: En su funcionalidad especial, debe mostrar un mensaje que diga "Cambiando de emisora".

Ejercicio- Combinar interfaces y clases abstractas

- Crea una **interfaz** llamada OperacionesBancarias que tenga los siguientes métodos:
 - depositar(cantidad: number): void
 - retirar(cantidad: number): void
- Crea una **clase abstracta** llamada CuentaBancaria que implemente la interfaz OperacionesBancarias y tenga:
 - Propiedades:
 - saldo: number (inicializado a 0)
 - Métodos:
 - depositar(cantidad: number): void:
 - retirar(cantidad: number): void:
 - Método abstracto tipoDeCuenta():
 - Crea dos clases concretas que hereden de CuentaBancaria:
- CuentaCorriente: En tipoDeCuenta() debe devolver "Cuenta Corriente".
- CuentaDeAhorros: En tipoDeCuenta() debe devolver "Cuenta de Ahorros".