Prog. Orientada a Objetos

Programador full-stack

Repaso General de POO

Agenda

- Organización de una Clase
- Composición
- Herencia
- Interfaces
- Clases Abstractas
- Manejo de Errores
- Recomendaciones Generales
- Ejercicios

Organización de una Clase

Variables Internas

Constructor

Métodos Públicos

Métodos Privados

```
export default class Recta {
  private puntoA: Punto;
  private puntoB: Punto;

  public constructor(puntoA: Punto, puntoB: Punto) {
    this.asignarValores();
  }

  public getPuntoA(): Punto {
    return this.puntoA;
  }

  public getPuntoB(): Punto {
    return this.puntoB;
  }

  private asignarValores(puntoA: Punto, puntoB: Punto) {
    this.puntoA = puntoA;
    this.puntoB = puntoB;
  }
}
```

Este es el orden que siempre se tiene que respetar para que la clase sea legible

Composición

- Se usa cuando queremos que una clase compleja esté *compuesta* por clases más sencillas
  - Un Auto compuesto por Motor, Rueda, Puerta, etc.
  - Un Televisor compuesto por Botón, Pantalla, etc.
  - Un Telefono compuesto por Pantalla, Cámara, Altavoz, Botón, etc.
  - Una clase Biblioteca compuesta por las clases Libro, Cliente, etc.
- Básicamente una composición es que una clase sencilla figure como *variable interna* de otra clase

Herencia

- Se usa cuando queremos que una clase haga lo mismo que otra, pero *agregando/modificando* funcionalidad
  - Un AutoDeportivo hace lo mismo que Auto, pero modifica la forma en que acelera
  - Un SmartTV hace lo mismo que un televisor común, pero agrega funciones como por ejemplo la conexión a internet
- Tener en cuenta el modificador *protected*
  - Es como el *private*, pero haciendo que las subclases puedan verlo

Interfaces

- Funcionan como un “contrato” que debe cumplir una determinada clase
- Se especifican los métodos pero sin comportamiento, es decir sin codificar lo que hacen
- Las clases que *implementen* a las interfaces tienen la obligación de implementar todos los métodos definidos en la interfaz
- Son una muy buena práctica ya que ayudan a plantear lo que debería hacer una clase

Clases Abstractas

- Son un punto intermedio entre las clases comunes y las interfaces.
- Requieren que al menos un método se especifique como abstracto, es decir que no esté implementado.
- Las clases abstractas no pueden ser instanciadas y por lo tanto deben ser heredadas por otra clase que de esta manera adquiere la obligación de implementar el o los métodos abstractos de su clase madre.
- Son una solución a casos de modelado de entidades similares pero con comportamiento diferente.

Manejo de Errores

- Cuando estamos desarrollando, la mayoría de las veces llamamos a métodos con los valores equivocados
- TypeScript provee mecanismos específicos para gestionar valores inválidos → *errores*
- Usando bloques *try/catch* podemos capturar errores esperados, para darle un tratamiento específico → permite que nuestro programa se recupere de los errores
- Usando *throw* podemos lanzar un determinado error, en caso de haberlo descubierto
- También se pueden definir errores propios
  - Se definen como clases que *extienden* de Error

Recomendaciones Generales

- Hacer *siempre* un planteo del sistema que vamos a hacer → diagrama de clase, interfaces, etc.
  - Al momento del planteo, pensar en función de cómo se debiera usar una clase *desde afuera* → esto permite ayudarnos a saber qué cosas hacerlas públicas y qué cosas no
- Una clase → una responsabilidad
  - Ejemplo: la clase Auto no puede tener un método que se encargue de calcular el promedio de un arreglo
- Usar nombres descriptivos
- Respetar la organización de una clase
- Tener paciencia con los conceptos, es imposible hacer las cosas bien de entrada → madurez

Evaluación del Módulo

- Va a consistir en tres partes
- Pensar en Objetos
  - A partir de una serie de requerimientos, implementar un sistema en TypeScript → prestar atención a encarar bien la solución, no hace falta preocuparse por detalles de implementación
- Documentación
  - A partir de un código en TypeScript, plantear el diagrama de clases → prestar atención a las relaciones entre las clases
- Preguntas Teóricas
  - Van a ser básicas pero con la idea de que puedan desarrollar y demostrar que conocen la teoría

Prog. Orientada a Objetos

Programador full-stack

Ejercicios

Parte 1 - Ejercicios

Diseño de Sistemas

- Se dispone de información referida a pistas de audio. Esta contiene un identificador, un título, la duración y el intérprete de diversas canciones.
- Se requiere implementar un sistema de administración de las mismas que permita armar listas de reproducción, teniendo que informar tanto la cantidad de pistas como la duración total de cada una de estas, como uno de sus servicios.
- Se deben realizar diagramas de clases y codificar utilizando los conceptos de POO. Considerar la utilizacion de una interface.

Parte 2 - Ejercicios

Partiendo del siguiente código:

```
interface PaymentMethod {
   pay(cost: number): void;
}

class Efectivo implements PaymentMethod {
    public pay(costo: number): void {
        console.log('Se pagó ' + costo + ' empleando efectivo');
    }
}

class Tarjeta implements PaymentMethod {
    public pay(costo: number): void {
        console.log('Se pagó ' + costo + ' empleando tarjeta');
    }
}

class Item {
    private descripcion: string;
    private costo: number;

    public constructor(descripcion: string, costo: number) {
        this.descripcion = descripcion;
        this.costo = costo;
    }

    public getDescripcion(): string { return this.descripcion; }
    public getCosto(): number { return this.costo; }
    public setDescripcion(descripcion: string): void { this.descripcion = descripcion; }
    public setCosto(costo: number): void { this.costo = costo; }

    public equals(i: Item): boolean {
         return this.costo == i.getCosto() && this.descripcion == i.getDescripcion();
    }
}
```

```
class Cuenta {
    private lineItems: Item[] = [];


    public addLineItem(lineItem: Item): void {
        this.lineItems.push(lineItem);
    }

    public removeLineItem(lineItem: Item): void {
        for (let i = 0; i < this.lineItems.length; i++) {
            if (this.lineItems[i].equals(lineItem))
                this.lineItems.splice(i, 1);
        }
    }

    public getCostInCents(): number {
        return this.lineItems
                        .map(item => item.getCosto())
                        .reduce((a, b) => a + b, 0);
    }

    public pay(method: PaymentMethod): void {
        method.pay(this.getCostInCents());
    }
}
```

Parte 2 - Ejercicios

- Generar diagrama de clases usando draw.io
- Escribir un comentario de cada uno de los métodos implementados → el objetivo es ir ejercitando la justificación de las decisiones de diseño tomadas
- Con respecto a las clases, escribir un comentario para cada una, indicando la responsabilidad de cada clase, y la funcionalidad que provee

Parte 3 - Preguntas Teóricas

- Responder las preguntas en un TXT y subirlo a GitHub
  - Enumerar tres funcionalidades de NPM y describirlas
  - ¿Cuál es el beneficio de usar un lenguaje con tipos?
  - ¿A qué se le llama variable interna? ¿Por qué internas?
  - Explicar la diferencia entre composición y herencia
  - Explicar el mecanismo que provee TypeScript para manejar casos en donde los parámetros que le llegan a un método son inválidos