EXERCISE- 1

1. Behavioral Design Patterns

Behavioral design patterns focus on communication between objects.

Use Case 1: Observer Pattern (Stock Price Tracker)

Description: The Observer pattern is used when an object (subject) has to notify other objects (observers) of changes without needing to know who or how many observers exist.

Example: A stock price tracker where multiple traders (observers) are notified when the price of a stock (subject) changes.

```
// Observer Pattern

// Subject (Stock)
class Stock {
  private observers: Observer[] = [];
  private price: number;

  constructor(public name: string, price: number) {
    this.price = price;
  }

  public attach(observer: Observer): void {
    this.observers.push(observer);
  }

  public setPrice(price: number): void {
    this.price = price;
    this.notifyAll();
  }

  private notifyAll(): void {
    this.observers.forEach(observer => observer.update(this.price));
  }
}

// Observer (Trader)
interface Observer {
  update(price: number): void;
}

class Trader implements Observer {
  constructor(public name: string) {}

  update(price: number): void {
    console.log(`${this.name} is notified. New stock price: $${price}`);
  }
}

// Usage
const googleStock = new Stock('Google', 1500);
const trader1 = new Trader('Trader 1');
const trader2 = new Trader('Trader 2');
```

```
googleStock.attach(trader1);
googleStock.attach(trader2);

googleStock.setPrice(1550);
```

Use Case 2: Command Pattern (Remote Control)

Description: The Command pattern encapsulates a request as an object, thereby allowing parameterization of clients with queues, requests, and operations.

Example: A remote control with multiple buttons that can trigger commands such as turning on/off the light and fan.

```
// Command Pattern

// Command Interface
interface Command {
  execute(): void;
}

// Receiver (Light)
class Light {
  turnOn(): void {
    console.log('Light is ON');
  }

  turnOff(): void {
    console.log('Light is OFF');
  }
}

// Concrete Command (Light On Command)
class LightOnCommand implements Command {
  private light: Light;

  constructor(light: Light) {
    this.light = light;
  }

  execute(): void {
    this.light.turnOn();
  }
}

// Concrete Command (Light Off Command)
class LightOffCommand implements Command {
  private light: Light;

  constructor(light: Light) {
    this.light = light;
  }

  execute(): void {
    this.light.turnOff();
  }
}

// Invoker (Remote Control)
class RemoteControl {
  private command: Command;

  setCommand(command: Command): void {
    this.command = command;
  }

  pressButton(): void {
    this.command.execute();
  }
}

// Usage
const light = new Light();
const lightOn = new LightOnCommand(light);
const lightOff = new LightOffCommand(light);

const remote = new RemoteControl();

remote.setCommand(lightOn);
remote.pressButton(); // Light is ON
remote.setCommand(lightOff);
remote.pressButton(); // Light is OFF
```

2. Creational Design Patterns

Creational design patterns focus on how objects are created.

Use Case 1: Factory Pattern (Car Factory)

Description: The Factory pattern provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Example: A car factory where different types of cars (SUV, Sedan) are created based on user input.

```
// Factory Pattern

// Product Interface (Car)
interface Car {
  drive(): void;
}

// Concrete Products (SUV and Sedan)
class SUV implements Car {
  drive(): void {
    console.log('Driving an SUV');
  }
}

class Sedan implements Car {
  drive(): void {
    console.log('Driving a Sedan');
  }
}

// Creator (Car Factory)
class CarFactory {
  static createCar(type: string): Car {
    if (type === 'SUV') {
      return new SUV();
    } else if (type === 'Sedan') {
      return new Sedan();
    } else {
      throw new Error('Invalid car type');
    }
  }
}

// Usage
const myCar = CarFactory.createCar('SUV');
myCar.drive(); // Driving an SUV
```

Use Case 2: Singleton Pattern (Logger Service)

Description: The Singleton pattern ensures a class has only one instance and provides a global point of access to it.

Example: A logger service where only one instance of the logger can be created to log system messages.

```
// Singleton Pattern

class Logger {
  private static instance: Logger;

  private constructor() {} // Private constructor prevents instantiation from outside

  public static getInstance(): Logger {
    if (!Logger.instance) {
      Logger.instance = new Logger();
    }
    return Logger.instance;
  }

  log(message: string): void {
    console.log(`Log message: ${message}`);
  }
}

// Usage
const logger1 = Logger.getInstance();
const logger2 = Logger.getInstance();

logger1.log('Singleton works!'); // Log message: Singleton works!

console.log(logger1 === logger2); // true (same instance)
```

3. Structural Design Patterns

Structural design patterns deal with object composition and simplify the design by identifying simple ways to combine objects.

Use Case 1: Adapter Pattern (Round Peg and Square Peg)

Description: The Adapter pattern allows incompatible interfaces to work together.

Example: A round peg that needs to fit into a square hole using an adapter.

```
// Adapter Pattern

// Target Interface (RoundHole)
class RoundHole {
  constructor(public radius: number) {}

  fits(peg: RoundPeg): boolean {
    return peg.radius <= this.radius;
  }
}

// Adaptee (SquarePeg)
class SquarePeg {
  constructor(public width: number) {}
}

// Adapter (SquarePegAdapter)
class SquarePegAdapter extends RoundPeg {
  private squarePeg: SquarePeg;

  constructor(squarePeg: SquarePeg) {
    super(squarePeg.width * Math.sqrt(2) / 2); // Approximation
    this.squarePeg = squarePeg;
  }
}

// Usage
const roundHole = new RoundHole(5);
const roundPeg = new RoundPeg(5);
console.log(roundHole.fits(roundPeg)); // true

const smallSquarePeg = new SquarePeg(5);
const squarePegAdapter = new SquarePegAdapter(smallSquarePeg);
console.log(roundHole.fits(squarePegAdapter)); // true (thanks to adapter)
```

Use Case 2: Decorator Pattern (Coffee Shop)

Description: The Decorator pattern attaches additional responsibilities to an object dynamically. It's a flexible alternative to subclassing for extending functionality.

Example: A coffee shop where customers can add extra ingredients (like milk or sugar) to their coffee.

```
// Decorator Pattern
// Component (Coffee)
interface Coffee {
  cost(): number;
  description(): string;
}

// Concrete Component (Simple Coffee)
class SimpleCoffee implements Coffee {
  cost(): number {
    return 5;
  }

  description(): string {
    return 'Simple Coffee';
  }
}

// Decorator (Coffee Decorator)
class CoffeeDecorator implements Coffee {
  protected coffee: Coffee;

  constructor(coffee: Coffee) {
    this.coffee = coffee;
  }

  cost(): number {
    return this.coffee.cost();
  }

  description(): string {
    return this.coffee.description();
  }
}

// Concrete Decorators
class MilkDecorator extends CoffeeDecorator {
  cost(): number {
    return this.coffee.cost() + 2;
  }

  description(): string {
    return this.coffee.description() + ', Milk';
  }
}

class SugarDecorator extends CoffeeDecorator {
  cost(): number {
    return this.coffee.cost() + 1;
  }

  description(): string {
    return this.coffee.description() + ', Sugar';
  }
}

// Usage
let myCoffee: Coffee = new SimpleCoffee();
console.log(`${myCoffee.description()}: $${myCoffee.cost()}`);

myCoffee = new MilkDecorator(myCoffee);
console.log(`${myCoffee.description()}: $${myCoffee.cost()}`);

myCoffee = new SugarDecorator(myCoffee);
console.log(`${myCoffee.description()}: $${myCoffee.cost()}`);
```

Conclusion

- **Behavioral**:
  - *Observer Pattern*: Stock price tracker.
  - *Command Pattern*: Remote control for light.
- **Creational**:
  - *Factory Pattern*: Car factory.
  - *Singleton Pattern*: Logger service.
- **Structural**:
  - *Adapter Pattern*: Round peg and square peg.
  - *Decorator Pattern*: Coffee shop with extra ingredients.

Each pattern is showcased with a practical example and the above full TypeScript code can be executed as a single file within a Node.js environment or TypeScript setup.

The above given codes are the executed once