Design Pattern

The Singleton design pattern ensures that a class has only one instance and provides a global point of access to it. This pattern is commonly used for managing shared resources, such as configuration settings or connection pools.

By Eslam Khder


```java
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        // Private constructor to prevent instantiation
    }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

By Eslam Khder

Factory Design Pattern

The Factory Design Pattern is a creational design pattern used to create objects without specifying the exact class of object that will be created. This pattern provides a way to delegate the instantiation logic to subclasses, promoting loose coupling and adherence to the Open/Closed Principle.

By Eslam Khder


```java
// Product interface
interface Shape {
    void draw();
}

// Concrete Products
class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}

class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Square");
    }
}
```

By Eslam Khder


```java
// Creator
class ShapeFactory {
    public static Shape createShape(String shapeType) {
        if (shapeType.equalsIgnoreCase("Circle")) {
            return new Circle();
        } else if (shapeType.equalsIgnoreCase("Square")) {
            return new Square();
        } else {
            throw new IllegalArgumentException("Unknown shape type");
        }
    }
}

// Client code
public class FactoryPatternDemo {
    public static void main(String[] args) {
        Shape shape1 = ShapeFactory.createShape("Circle");
        shape1.draw();  // Output: Drawing a Circle

        Shape shape2 = ShapeFactory.createShape("Square");
        shape2.draw();  // Output: Drawing a Square
    }
}
```

By Eslam Khder

Dependency Injection (DI) - Inversion of Control (IoC)

Dependency Injection (DI) is a design pattern used to implement Inversion of Control (IoC), allowing for more modular, testable, and maintainable code. In DI, an object (often called a "client") receives its dependencies from an external source rather than creating them itself. This makes it easier to manage dependencies and promotes loose coupling.

By Eslam Khder