

1. Singleton Pattern

Ensures that a class has only **one instance** and provides a global access point to it.

```
class Singleton {  
    private static Singleton instance;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
  
    public void showMessage() {  
        System.out.println("Singleton Instance");  
    }  
}  
  
public class SingletonDemo {  
    public static void main(String[] args) {  
        Singleton obj = Singleton.getInstance();  
        obj.showMessage();  
    }  
}
```

Don't forget to check the description- How to make this 100% Singleton?

2. Factory Pattern

Provides an interface for creating objects, but allows **subclasses to decide which class to instantiate**.

```
interface Shape {  
    void draw();  
}  
  
class Circle implements Shape {  
    public void draw() {  
        System.out.println("Drawing a Circle");  
    }  
}  
  
class Rectangle implements Shape {  
    public void draw() {  
        System.out.println("Drawing a Rectangle");  
    }  
}  
  
class ShapeFactory {  
    public static Shape getShape(String type) {  
        if (type.equalsIgnoreCase("CIRCLE")) {  
            return new Circle();  
        } else if (type.equalsIgnoreCase("RECTANGLE")) {  
            return new Rectangle();  
        }  
        return null;  
    }  
}  
  
public class FactoryDemo {  
    public static void main(String[] args) {  
        Shape shape1 = ShapeFactory.getShape("CIRCLE");  
        shape1.draw();  
  
        Shape shape2 = ShapeFactory.getShape("RECTANGLE");  
        shape2.draw();  
    }  
}
```

}

}

3. Abstract Factory Pattern

Creates families of related objects without specifying their **concrete classes**.

```
interface Animal {  
    void makeSound();  
}  
  
class Dog implements Animal {  
    public void makeSound() {  
        System.out.println("Bark");  
    }  
}  
  
class Cat implements Animal {  
    public void makeSound() {  
        System.out.println("Meow");  
    }  
}  
  
abstract class AnimalFactory {  
    abstract Animal createAnimal();  
}  
  
class DogFactory extends AnimalFactory {  
    public Animal createAnimal() {  
        return new Dog();  
    }  
}  
  
class CatFactory extends AnimalFactory {  
    public Animal createAnimal() {  
        return new Cat();  
    }  
}  
  
public class AbstractFactoryDemo {  
    public static void main(String[] args) {  
        AnimalFactory dogFactory = new DogFactory();
```

```
        Animal dog = dogFactory.createAnimal();
        dog.makeSound();

        AnimalFactory catFactory = new CatFactory();
        Animal cat = catFactory.createAnimal();
        cat.makeSound();
    }
}
```

4. Builder Pattern

Used to construct **complex objects step by step**.

```
class Car {  
    private String engine;  
    private int wheels;  
  
    private Car(CarBuilder builder) {  
        this.engine = builder.engine;  
        this.wheels = builder.wheels;  
    }  
  
    public static class CarBuilder {  
        private String engine;  
        private int wheels;  
  
        public CarBuilder setEngine(String engine) {  
            this.engine = engine;  
            return this;  
        }  
  
        public CarBuilder setWheels(int wheels) {  
            this.wheels = wheels;  
            return this;  
        }  
  
        public Car build() {  
            return new Car(this);  
        }  
    }  
  
    public void showCar() {  
        System.out.println("Car with Engine: " + engine + ", Wheels:  
" + wheels);  
    }  
}  
  
public class BuilderDemo {  
    public static void main(String[] args) {
```

```
    Car car = new
Car.CarBuilder().setEngine("V8").setWheels(4).build();
    car.showCar();
}
}
```

5. Prototype Pattern

Creates **new objects by copying an existing object**, reducing the overhead of creating complex objects.

```
import java.util.HashMap;
import java.util.Map;

abstract class Animal implements Cloneable {
    public String name;

    public abstract void makeSound();

    public Animal clone() throws CloneNotSupportedException {
        return (Animal) super.clone();
    }
}

class Sheep extends Animal {
    public Sheep() {
        this.name = "Sheep";
    }

    public void makeSound() {
        System.out.println("Baa Baa");
    }
}

class PrototypeRegistry {
    private static Map<String, Animal> registry = new HashMap<>();

    static {
        registry.put("Sheep", new Sheep());
    }

    public static Animal getClone(String type) throws
CloneNotSupportedException {
        return registry.get(type).clone();
    }
}
```

```
}

public class PrototypeDemo {
    public static void main(String[] args) throws
CloneNotSupportedException {
    Animal clonedSheep = PrototypeRegistry.getClone("Sheep");
    clonedSheep.makeSound();
}
}
```
