The **Factory Design Pattern** is a **creational design pattern** that provides a way to create objects **without exposing the instantiation logic to the client**. Instead of using the `new` keyword directly, a factory method is used to create the object.

🏭 Simple Definition:

Factory Pattern creates objects based on input or configuration, **hiding the object creation logic** from the client code.

🔧 Real-life Analogy:

Imagine a **car factory**. You ask the factory for a car of a certain type (e.g., "SUV" or "Sedan"). You don't need to know how it's built—you just receive the correct car.

💡 When to Use:

- When you have a **common interface or superclass** but different implementations.
- When the client shouldn't need to know the concrete class name.
- When object creation is **complex** or **dependent on conditions**.

🧱 Structure:

java
CopyEdit

```java
// Step 1: Create an interface
interface Shape {
    void draw();
}

// Step 2: Create concrete classes
class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing Circle");
    }
}
```

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

// Step 3: Create Factory class
class ShapeFactory {
    public Shape getShape(String shapeType) {
        if (shapeType == null) return null;
        if (shapeType.equalsIgnoreCase("CIRCLE")) return new Circle();
        if (shapeType.equalsIgnoreCase("SQUARE")) return new Square();
        return null;
    }
}

// Step 4: Use the Factory
public class Main {
    public static void main(String[] args) {
        ShapeFactory factory = new ShapeFactory();

        Shape s1 = factory.getShape("CIRCLE");
        s1.draw(); // Output: Drawing Circle

        Shape s2 = factory.getShape("SQUARE");
        s2.draw(); // Output: Drawing Square
    }
}
```

✅ Benefits:

- Loose coupling between client and object creation.
- Easy to add new types without modifying the client code.
- Centralizes object creation logic.

❌ Drawbacks:

- Can lead to more classes and complexity.
- May require reflection or if-else/`switch` blocks if not carefully structured.

Would you like a real-world use case from Spring Framework where the factory pattern is applied?