Great choice again! Let's now walk through the **Factory Design Pattern** using the same structured, example-first approach: **problem**, **pattern-based solution**, **class diagram**, **runtime behavior**, and **pros/cons**.

🏪 Real-World Example: Notification System

You're building a system that can send different types of notifications:

- **Email**
- **SMS**
- **Push Notification**

Each notification type has its own configuration and behavior.

❌ Naive Implementation (Problem)

You might start with something like this:

```
public class NotificationService {
    public void sendNotification(String type) {
        if (type.equals("EMAIL")) {
            new EmailNotification().send();
        } else if (type.equals("SMS")) {
            new SMSNotification().send();
        } else if (type.equals("PUSH")) {
            new PushNotification().send();
        }
    }
}
```

😖 Problems:

- 🔗 **Tightly coupled** to concrete classes
- 📏 Violates **Open/Closed Principle**
- 📦 Hard to manage and scale
- 🔧 Changes in object creation require changes in logic

✅ Solution: Factory Design Pattern

Instead of creating objects in the `NotificationService`, we **delegate creation** to a **factory class**.

The Factory Pattern encapsulates object creation and returns a **common interface**, so the rest of the system can stay agnostic to the concrete class.

🧠 Pattern Breakdown

🔹 Step 1: Define Common Interface

```
public interface Notification {
    void send();
}
```

🔹 Step 2: Create Concrete Notification Types

```
public class EmailNotification implements Notification {
    public void send() {
        System.out.println("Sending Email Notification");
    }
}

public class SMSNotification implements Notification {
    public void send() {
        System.out.println("Sending SMS Notification");
    }
}

public class PushNotification implements Notification {
    public void send() {
        System.out.println("Sending Push Notification");
    }
}
```

🔹 Step 3: Factory Class to Create Notifications

```
public class NotificationFactory {
    public static Notification createNotification(String type) {
        switch (type.toUpperCase()) {
            case "EMAIL": return new EmailNotification();
            case "SMS": return new SMSNotification();
            case "PUSH": return new PushNotification();
            default: throw new IllegalArgumentException("Invalid type");
        }
    }
}
```

🔹 Step 4: Use Factory in Client Code

```
Notification notification =
NotificationFactory.createNotification("EMAIL");
notification.send(); // Output: Sending Email Notification
```

🏗️ Class Diagram

```
           +-----------------+
           |  Notification   | ←── Interface
           +-----------------+
           | + send()        |
           +-----------------+
              ▲     ▲     ▲
              |     |     |
     +------------+ +-------------+ +-----------------+
     |   Email    | |     SMS     | |       Push      |
     +------------+ +-------------+ +-----------------+
     | + send()   | | + send()    | |    + send()     |
     +------------+ +-------------+ +-----------------+
                          ▲
                          |
         +-------------------------------+
         |       NotificationFactory     |
         +-------------------------------+
         | + createNotification(String)  |
         +-------------------------------+
```

🧪 Runtime Usage

```
Notification n1 = NotificationFactory.createNotification("SMS");
n1.send(); // Sending SMS Notification

Notification n2 = NotificationFactory.createNotification("PUSH");
n2.send(); // Sending Push Notification
```

✅ Benefits Gained

Benefit	Explanation
✅ Centralized object creation	All instantiations handled in one place
✅ Loose coupling	Client code depends only on the interface
✅ Scalable	Add new types with minimal changes
✅ Open/Closed Principle	Factory grows without modifying clients

❌ Drawbacks

Drawback	Solution
❌ Uses conditionals	Can replace with Reflection or Factory Registry
❌ One factory for many objects may grow big	Use Abstract Factory or split factories by category

Drawback	Solution
❌ Logic spread in `switch-case`	Use `Map<String, Supplier<Notification>>` for extensibility

📦 Summary Table

Aspect	Description
Pattern Name	Factory
Intent	Delegate object creation to a separate component
Use Cases	Notification system, shape creation, parsers, DAOs
Key Benefit	Decouples object creation from usage
Core Principle	Encapsulation of instantiation logic

Would you like a **visual illustration of how the factory returns different objects at runtime** (like a vending machine analogy)?

Perfect! You're asking for a more **scalable and extensible version** of the Factory Pattern — often called the **Registry-based Factory**.

This approach **avoids switch-cases** entirely and allows each implementation to **self-register** to a factory map, making it **Open/Closed Principle** compliant.

✅ Advanced Scalable Factory: Registry-Based Implementation

🎯 Real-World Case: Notification System

We'll allow **new notification types to register themselves** with the factory.

This allows your factory to stay **unchanged**, no matter how many new types you add later.

🧠 Step-by-Step Implementation

🔷 Step 1: Common Interface

```
public interface Notification {
    void send();
}
```

🔷 Step 2: Factory with Registry Map

```
public class NotificationFactory {
    private static final Map<String, Supplier<Notification>> registry = new
HashMap<>();

    public static void register(String type, Supplier<Notification>
supplier) {
        registry.put(type.toUpperCase(), supplier);
    }

    public static Notification createNotification(String type) {
        Supplier<Notification> supplier = registry.get(type.toUpperCase());
        if (supplier == null) {
            throw new IllegalArgumentException("No such notification type
registered: " + type);
        }
        return supplier.get();
    }
```

```
}
```

🔷 Step 3: Concrete Implementations Self-Register

```
public class EmailNotification implements Notification {
    static {
        NotificationFactory.register("EMAIL", EmailNotification::new);
    }

    public void send() {
        System.out.println("Sending Email Notification");
    }
}

public class SMSNotification implements Notification {
    static {
        NotificationFactory.register("SMS", SMSNotification::new);
    }

    public void send() {
        System.out.println("Sending SMS Notification");
    }
}

public class PushNotification implements Notification {
    static {
        NotificationFactory.register("PUSH", PushNotification::new);
    }

    public void send() {
        System.out.println("Sending Push Notification");
    }
}
```

🔷 Step 4: Usage

You just need to ensure classes are loaded (e.g., force-load them once via `Class.forName` or load all from a package dynamically).

```
public class Client {
    public static void main(String[] args) {
        // Optionally force class loading
        try {
            Class.forName("EmailNotification");
            Class.forName("SMSNotification");
            Class.forName("PushNotification");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        Notification notif =
NotificationFactory.createNotification("email");
        notif.send(); // Sending Email Notification

        Notification notif2 =
NotificationFactory.createNotification("push");
```

```
        notif2.send(); // Sending Push Notification
    }
}
```

🏗️ Updated Class Diagram

```
                     +----------------------+
                     |    Notification      |   (Interface)
                     +----------------------+
                                 ▲

+--------------------+ +--------------------+ +---------------------+
| EmailNotification| | SMSNotification    | | PushNotification    |
+--------------------+ +--------------------+ +---------------------+
| +send()           | | +send()            | | +send()             |
| static register()| | static register()| | static register()   |
+--------------------+ +--------------------+ +---------------------+

                     +---------------------------+
                     |    NotificationFactory    |
                     +---------------------------+
                     | - Map<String, Supplier>   |
                     | + register(type, sup)     |
                     | + createNotification()    |
                     +---------------------------+
```

✅ Benefits of This Approach

Benefit	Explanation
✅ Extensible	Add new types without touching the factory
✅ Clean & DRY	Avoids switch-cases and if-else chains
✅ Open/Closed Principle	Fully respected — factory never changes
✅ Decoupled	Factory doesn't know concrete implementations
✅ Runtime Dynamic	New notification types can be loaded via config or plugins

---

❌ Potential Drawbacks

Drawback	Solution
⚠️ Class loading required	Force load via reflection or Spring-like scanning
⚠️ Harder to trace control flow	Use logging during registration
⚠️ Registry may become global	Can encapsulate it within a DI container

---

📦 Summary

Aspect	Value
Pattern	Registry-based Factory
Use Case	Dynamically load and register object creators
Best For	Plugin-based systems, extensible modules, scalable object creation
Core Idea	Delegate instantiation to registered suppliers in a map

---

Would you like a **visual image of this registration mechanism**, showing how each class self-registers into the central factory?