Factory method level 1

Factory Method

now we go level by level growth like a game on this java topic, provide coding example with its explanation, real-world use case example,necessary hints, points to remember,usual mistakes,etc.

start lvl1

Great choice 🚀!
We'll master **Factory Method Pattern** step by step, like a game 🎮.

🎯 Level 1: Introduction to Factory Method Pattern

🔑 Definition

The **Factory Method Pattern** is a **creational design pattern** that provides an interface for creating objects but lets subclasses decide which class to instantiate.

Instead of calling a constructor directly (`new` keyword everywhere), we delegate the creation logic to a **factory method**.

💡 Why do we need it?

- Avoids tight coupling between code and specific classes.
- Makes code more **flexible** and **scalable** when new product types are introduced.
- Centralizes object creation → Easier to maintain.

⚡ Simple Example (Without Factory)

Suppose we want to create different **shapes** like Circle and Rectangle.

```
java

// Product classes
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");
    }
}

// Main usage without Factory
public class WithoutFactory {
    public static void main(String[] args) {
        Shape shape1 = new Circle();    // directly using 'new'
        Shape shape2 = new Rectangle();

        shape1.draw();
        shape2.draw();
    }
}
```

👉 Problem: Every time we add a new **`Shape`** (like Triangle, Polygon), we need to change the main code. This creates **tight coupling**.

⚡ With Factory Method (Level 1)

We introduce a **Factory** that decides which object to create.

```
java

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

// Concrete Products
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");
    }
}

// Factory (decides which object to create)
class ShapeFactory {
    public Shape getShape(String type) {
        if (type.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        } else if (type.equalsIgnoreCase("RECTANGLE")) {
            return new Rectangle();
        }
        return null;
    }
}

// Main
public class FactoryPatternDemo {
    public static void main(String[] args) {
        ShapeFactory factory = new ShapeFactory();

        Shape shape1 = factory.getShape("CIRCLE");
        Shape shape2 = factory.getShape("RECTANGLE");

        shape1.draw();
        shape2.draw();
    }
}
```

🏗️ How this helps?

- Now if we add `Triangle`, we only modify `ShapeFactory`, not the main code.
- Main class doesn't depend on concrete classes → **Loose Coupling**.

🌍 Real-World Example (Level 1)

- **Document creation software**: Word processors can create `WordDocument`, `PdfDocument`, or `ExcelDocument`. Instead of directly using `new WordDocument()`, a **DocumentFactory** decides which to create.
- **Database drivers**: JDBC `DriverManager.getConnection(url)` is a factory method. Depending on the URL, it decides whether to create a MySQL connection, Oracle connection, etc.

📝 Hints & Points to Remember

- **Factory Method** is about delegating object creation.
- Reduces `new` keyword usage in main logic.
- Improves **extensibility** (add new product with less changes).

⚠️ Usual Mistakes

1. Making **too many factories** → leads to complexity.
2. Forgetting to use **interfaces/abstract classes** → loses flexibility.
3. Treating Factory Method = Abstract Factory (they are different, we'll see later).

👉 Level 1 is complete ✅ (basic example, intro, and real-world use).

Would you like me to take you to **Level 2** → where we make it more **object-oriented** using **abstract factory methods inside classes** instead of simple if-else?

next

Perfect 🎯!
Now we step into **Level 2** of the Factory Method journey 🚀

🎮 Level 2: Factory Method with Abstract Creator

🔑 Core Idea

In **Level 1**, we had a simple `ShapeFactory` with `if-else` deciding what to create.
👉 But that's not *true* Factory Method — it's closer to a **Simple Factory**.

In the **Factory Method Pattern**,

- We **delegate object creation to subclasses**,
- The **factory method is abstract** in a base class,
- Each **concrete subclass** decides which object to return.

⚡ Coding Example

Step 1: Product

```
java

// Product interface
interface Transport {
    void deliver();
}
```

Step 2: Concrete Products

```
java

class Truck implements Transport {
    public void deliver() {
        System.out.println("Delivering by land in a truck.");
    }
}

class Ship implements Transport {
    public void deliver() {
        System.out.println("Delivering by sea in a ship.");
    }
}
```

Step 3: Creator (Abstract Factory Class)

```
java

abstract class Logistics {
    // Factory Method (abstract)
    public abstract Transport createTransport();

    // Business logic that uses the product
    public void planDelivery() {
        Transport t = createTransport(); // object created by factory method
        t.deliver();
    }
}
```

Step 4: Concrete Factories

```java
class RoadLogistics extends Logistics {
    public Transport createTransport() {
        return new Truck();
    }
}

class SeaLogistics extends Logistics {
    public Transport createTransport() {
        return new Ship();
    }
}
```

Step 5: Client

```java
public class FactoryMethodDemo {
    public static void main(String[] args) {
        Logistics road = new RoadLogistics();
        road.planDelivery();   // uses Truck internally

        Logistics sea = new SeaLogistics();
        sea.planDelivery();    // uses Ship internally
    }
}
```

🔍 Explanation

- `Logistics` = **Abstract Creator** (declares factory method `createTransport`)
- `RoadLogistics` & `SeaLogistics` = **Concrete Creators** (decide which product to instantiate)
- `Truck` & `Ship` = **Products**

The **client code (`FactoryMethodDemo`) never knows which transport object is created.**
It just asks `planDelivery()` → factory method decides.

🌍 Real-World Use Case

- **GUI Toolkits**:
  - Windows OS returns `WindowsButton`, MacOS returns `MacButton`.
  - The app just calls `button.render()` — no need to know platform details.
- **Payment Systems**:
  - `PaymentProcessor` (abstract) has factory method `createPayment()`.
  - `CreditCardProcessor`, `UPIProcessor`, `PayPalProcessor` decide which payment object to return.

📝 Hints & Points to Remember

1. Use **abstract classes or interfaces** for products.
2. Factory Method = “subclass decides the product”.

3. Better for **scalability**: If new `AirLogistics` is added → no change in client code.

⚠️ Usual Mistakes

- ❌ Putting `if-else` in every factory class → you end up with Simple Factory again.
- ❌ Mixing Factory Method with Singleton (they solve different problems).
- ❌ Not using abstraction → makes factory meaningless.

✅ Level 2 complete!
Now we have a **true Factory Method pattern** with **abstract factory classes and polymorphism**.

Would you like me to take you to **Level 3** → where we compare **Simple Factory vs Factory Method vs Abstract Factory** with examples (so you don't confuse them in interviews)?

next

Awesome ⚡ Divakar, we're now entering **Level 3** of the Factory Method journey 🎮.
This is the **comparison level** — many developers (and even interviewers) confuse **Simple Factory**, **Factory Method**, and **Abstract Factory**.

Let's clear it up step by step with examples ✅

🎯 Level 3: Simple Factory vs Factory Method vs Abstract Factory

---

1️⃣ Simple Factory (Not a Design Pattern officially)

👉 One factory class decides which object to create using **if-else / switch**.

- Centralized creation
- But **not extensible** (must edit factory each time).

Example:

```
java

class ShapeFactory {
    public Shape getShape(String type) {
        if (type.equalsIgnoreCase("CIRCLE")) return new Circle();
        else if (type.equalsIgnoreCase("RECTANGLE")) return new Rectangle();
        return null;
    }
}
```

🔎 **Use when**: You need a single place for object creation, but don't expect frequent extensions.

2️⃣ Factory Method (A true Gang of Four Pattern ✅)

👉 Creation logic is **delegated to subclasses**.

- Parent defines **abstract factory method**.
- Subclasses decide what to instantiate.
- Uses **inheritance + polymorphism**.

Example:

```
java

abstract class Logistics {
    public abstract Transport createTransport(); // factory method
}

class RoadLogistics extends Logistics {
    public Transport createTransport() { return new Truck(); }
}

class SeaLogistics extends Logistics {
    public Transport createTransport() { return new Ship(); }
}
```

🔎 **Use when**: You want to follow **Open/Closed Principle** (easily add new types without changing existing code).

3️⃣ Abstract Factory (Super level 🌍)

👉 A factory of factories.

- Creates **families of related objects**.
- Ensures objects created together are **compatible**.

Example:

```
java

// Products
interface Button { void render(); }
interface Checkbox { void render(); }

// Concrete Products for Windows
class WindowsButton implements Button { public void render(){ System.out.println("Windows
Button"); } }
class WindowsCheckbox implements Checkbox { public void render(){ System.out.println("Windows
Checkbox"); } }

// Concrete Products for Mac
class MacButton implements Button { public void render(){ System.out.println("Mac Button"); } }
class MacCheckbox implements Checkbox { public void render(){ System.out.println("Mac
Checkbox"); } }

// Abstract Factory
interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}
```

```
// Concrete Factories
class WindowsFactory implements GUIFactory {
    public Button createButton(){ return new WindowsButton(); }
    public Checkbox createCheckbox(){ return new WindowsCheckbox(); }
}
class MacFactory implements GUIFactory {
    public Button createButton(){ return new MacButton(); }
    public Checkbox createCheckbox(){ return new MacCheckbox(); }
}

// Client
public class AbstractFactoryDemo {
    public static void main(String[] args) {
        GUIFactory factory = new WindowsFactory(); // could switch to MacFactory
        Button button = factory.createButton();
        Checkbox checkbox = factory.createCheckbox();

        button.render();
        checkbox.render();
    }
}
```

🔎 **Use when**: You want to create **related object families** (UI components, DB connectors, etc.) without worrying about compatibility.

📊 Quick Comparison Table

Feature	Simple Factory	Factory Method	Abstract Factory
Who decides object creation?	Single factory class (if-else)	Subclass via factory method	Factory of factories
Extensibility	Hard (modify factory)	Easy (add subclass)	Easy (add new factory)
Abstraction Level	Low	Medium	High
Best Use Case	Small apps, utility methods	Frameworks, scalable apps	Families of related products (cross-platform UI, DBs)

🌍 Real-World Analogy

- **Simple Factory** → Vending machine with all logic inside: "if you press Coke, give Coke; if Pepsi, give Pepsi."
- **Factory Method** → A franchise shop: Headquarters defines "how to serve a drink," but each shop (subclass) decides Coke or Pepsi.
- **Abstract Factory** → A restaurant: You order a **meal set** (burger + drink). The factory ensures both items are **compatible** (McDonald's set vs KFC set).

📝 Hints for Interviews

- **Simple Factory** is not in GoF patterns, but commonly used.
- **Factory Method** → when product creation varies by subclass.
- **Abstract Factory** → when creating related product families.

⚠️ Usual Mistakes

1. Confusing Factory Method with Abstract Factory (Factory Method → 1 product at a time, Abstract Factory → family of products).
2. Thinking Factory removes `new` completely (you still use `new` inside factories, not outside).
3. Over-engineering small apps with Abstract Factory when Simple Factory is enough.

✅ Level 3 complete!
Now you have a clear **battle map of factories** 🏆.

Do you want me to take you to **Level 4** → where we implement a **real-world project-style Factory Method** (like Payment Gateway, Logger, or Notification System)?

next

Perfect ⚡ Divakar, welcome to **Level 4** of our Factory Method adventure 🎮.
Now we'll **apply Factory Method in a real-world project-style scenario**.

🎯 Level 4: Real-World Project — Notification System

📝 Problem

We want a system that can send notifications by **Email**, **SMS**, or **Push Notification**.

- The client should not care *how* the notification is sent.
- Adding a new notification type (like WhatsApp) should not break existing code.

⚡ Implementation with Factory Method

Step 1: Product (Notification)

```java

// Product
interface Notification {
    void notifyUser(String message);
}
```

Step 2: Concrete Products

```java
class EmailNotification implements Notification {
    public void notifyUser(String message) {
        System.out.println("Sending EMAIL: " + message);
    }
}

class SMSNotification implements Notification {
    public void notifyUser(String message) {
        System.out.println("Sending SMS: " + message);
    }
}

class PushNotification implements Notification {
    public void notifyUser(String message) {
        System.out.println("Sending PUSH notification: " + message);
    }
}
```

Step 3: Creator (Abstract Factory Class)

```java
abstract class NotificationFactory {
    public abstract Notification createNotification();

    // Business method using product
    public void send(String message) {
        Notification notification = createNotification();
        notification.notifyUser(message);
    }
}
```

Step 4: Concrete Factories

```java
class EmailNotificationFactory extends NotificationFactory {
    public Notification createNotification() {
        return new EmailNotification();
    }
}

class SMSNotificationFactory extends NotificationFactory {
    public Notification createNotification() {
        return new SMSNotification();
    }
}

class PushNotificationFactory extends NotificationFactory {
    public Notification createNotification() {
        return new PushNotification();
    }
}
```

Step 5: Client

```java
public class NotificationDemo {
    public static void main(String[] args) {
        NotificationFactory emailFactory = new EmailNotificationFactory();
```

```
        emailFactory.send("Welcome to our service!");

        NotificationFactory smsFactory = new SMSNotificationFactory();
        smsFactory.send("Your OTP is 123456");

        NotificationFactory pushFactory = new PushNotificationFactory();
        pushFactory.send("You have a new friend request!");
    }
}
```

✅ Output

```
vbnet

Sending EMAIL: Welcome to our service!
Sending SMS: Your OTP is 123456
Sending PUSH notification: You have a new friend request!
```

🌍 Real-World Use Cases

- **Payment Gateways**: Factory decides whether to process via CreditCard, UPI, PayPal.
- **Logging Frameworks**: Factory creates ConsoleLogger, FileLogger, or DatabaseLogger.
- **Cloud Services**: Factory decides AWS, Azure, or GCP service objects at runtime.

📝 Hints & Points to Remember

- Factory Method centralizes **business logic + object creation**.
- Adding new notification type = Just add a new Factory + Product → no client changes (Open/Closed Principle).
- This structure is **extensible & scalable** for real applications.

⚠️ Usual Mistakes

- ❌ Putting `if-else` inside client → breaks Factory Method purpose.
- ❌ Using static factory methods everywhere → turns into **Simple Factory** (less extensible).
- ❌ Not using interface/abstract → client gets tightly coupled to concrete class.

✅ Level 4 complete!
Now you've built a **project-style Factory Method system**. 🎉

👉 Do you want me to take you to **Level 5** → where we explore **advanced variations** like **Parameterised Factory Method**, **Reflection-based Factory**, and **when to prefer Factory Method vs Builder vs Prototype**?

```
public class NotificationDemo {
    public static void main(String[] args) {
        NotificationFactory emailFactory = new EmailNotificationFactory();
        emailFactory.send("Welcome to our service!");

        NotificationFactory smsFactory = new SMSNotificationFactory();
        smsFactory.send("Your OTP is 123456");

        NotificationFactory pushFactory = new PushNotificationFactory();
        pushFactory.send("You have a new friend request!");
    }
}
```

but main class new obj creation

Great observation Divakar 👋!
You've spotted an **important pain point**:

In the **NotificationDemo**, we are still writing:

```java
NotificationFactory emailFactory = new EmailNotificationFactory();
NotificationFactory smsFactory = new SMSNotificationFactory();
NotificationFactory pushFactory = new PushNotificationFactory();
```

👉 That means the **client is still creating concrete factory objects with `new`.**
So although we've applied **Factory Method**, the client is partially coupled to the factories.

🎯 How to Fix?

We want the **client to be completely decoupled** from concrete factories.
There are **3 common approaches**:

✅ 1. Parameterized Factory Method (Better Encapsulation)

Instead of creating each factory manually, we create a **centralized factory provider**.

```java
// Factory Provider (avoids 'new' in client)
class FactoryProvider {
    public static NotificationFactory getFactory(String type) {
        if (type.equalsIgnoreCase("EMAIL")) {
            return new EmailNotificationFactory();
        } else if (type.equalsIgnoreCase("SMS")) {
            return new SMSNotificationFactory();
        } else if (type.equalsIgnoreCase("PUSH")) {
            return new PushNotificationFactory();
        }
        throw new IllegalArgumentException("Unknown notification type");
```

```
    }
}
```

Updated Client:

```
java

public class NotificationDemo {
    public static void main(String[] args) {
        NotificationFactory factory;

        factory = FactoryProvider.getFactory("EMAIL");
        factory.send("Welcome to our service!");

        factory = FactoryProvider.getFactory("SMS");
        factory.send("Your OTP is 123456");

        factory = FactoryProvider.getFactory("PUSH");
        factory.send("You have a new friend request!");
    }
}
```

✅ Now the client **does not know about** `new EmailNotificationFactory()`.
It only knows `FactoryProvider.getFactory("EMAIL")`.

✅ 2. Reflection-Based Factory (More Dynamic, Less Code)

We can avoid hardcoding **if-else** with **Java Reflection**:

```
java

class FactoryProvider {
    public static NotificationFactory getFactory(Class<? extends NotificationFactory>
factoryClass) {
        try {
            return factoryClass.getDeclaredConstructor().newInstance();
        } catch (Exception e) {
            throw new RuntimeException("Factory creation failed", e);
        }
    }
}
```

Updated Client:

```
java

public class NotificationDemo {
    public static void main(String[] args) {
        NotificationFactory emailFactory =
FactoryProvider.getFactory(EmailNotificationFactory.class);
        emailFactory.send("Welcome via Email!");

        NotificationFactory smsFactory =
FactoryProvider.getFactory(SMSNotificationFactory.class);
        smsFactory.send("Your OTP is 123456");

        NotificationFactory pushFactory =
FactoryProvider.getFactory(PushNotificationFactory.class);
        pushFactory.send("Push: You have a new message!");
```

```
    }
}
```

✅ This way, adding new factories doesn't require changing FactoryProvider logic.

✅ 3. Use Abstract Factory / Dependency Injection

In large projects, we combine **Factory Method + Abstract Factory** or use frameworks like **Spring** (IoC/DI) where the container creates the factories/objects for you.

Example in **Spring**:

```
java

@Autowired
private NotificationFactory notificationFactory;
```

👉 You never use `new` — the DI container injects the correct factory.

📌 Conclusion

- Your concern is correct: `new` in main still couples client to concrete factory.
- Best fix = use a **FactoryProvider** or **Dependency Injection**.
- This makes **client code fully decoupled and future-proof**.

Do you want me to show you a **Level 5 version** where we implement this Notification Factory with **parameterized factory provider** + show how adding a new type (like WhatsAppNotification) requires **zero changes in client code**?

> still this is modification , which vailates principle

💯 You are absolutely right Divakar — very sharp thinking 👍.
You're noticing that even with a **FactoryProvider**, we still need to **modify** it when we add a new notification type (like WhatsApp).

That **does violate the Open/Closed Principle (OCP)**:

> *Software entities should be open for extension, but closed for modification.*

So the question becomes:
👉 How do we design a **Factory Method–based system** where adding a new product doesn't require *any* changes in existing code?

🎯 Level 5: Truly OCP-Compliant Factory

✅ Problem with our current Factory

- When we add a new `Notification` type → we must modify `FactoryProvider` (if-else or switch).
- That's **not open/closed**.

✅ Solution 1: Reflection + Registration (Dynamic Factory Registry)

We let factories **register themselves** in a central registry, so the client never changes.

Step 1: Product

```
java

interface Notification {
    void notifyUser(String message);
}
```

Step 2: Concrete Products

```
java

class EmailNotification implements Notification {
    public void notifyUser(String message) {
        System.out.println("EMAIL: " + message);
    }
}

class SMSNotification implements Notification {
    public void notifyUser(String message) {
        System.out.println("SMS: " + message);
    }
}
```

Step 3: Factory Interface

```
java

interface NotificationFactory {
    Notification createNotification();
}
```

Step 4: Concrete Factories

```
java

class EmailNotificationFactory implements NotificationFactory {
    public Notification createNotification() {
        return new EmailNotification();
    }
}

class SMSNotificationFactory implements NotificationFactory {
    public Notification createNotification() {
```

```
        return new SMSNotification();
    }
}
```

Step 5: Factory Registry (No Modification Later)

```
java

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

class FactoryRegistry {
    private static final Map<String, NotificationFactory> registry = new HashMap<>();

    // Register factory dynamically
    public static void register(String key, NotificationFactory factory) {
        registry.put(key.toUpperCase(), factory);
    }

    // Get factory
    public static NotificationFactory getFactory(String key) {
        return registry.get(key.toUpperCase());
    }
}
```

Step 6: Client

```
java

public class NotificationDemo {
    public static void main(String[] args) {
        // FACTORIES REGISTER THEMSELVES
        FactoryRegistry.register("EMAIL", new EmailNotificationFactory());
        FactoryRegistry.register("SMS", new SMSNotificationFactory());

        // CLIENT NEVER MODIFIES
        NotificationFactory emailFactory = FactoryRegistry.getFactory("EMAIL");
        emailFactory.createNotification().notifyUser("Welcome via Email!");

        NotificationFactory smsFactory = FactoryRegistry.getFactory("SMS");
        smsFactory.createNotification().notifyUser("Your OTP is 123456");
    }
}
```

✅ Now if we add WhatsAppNotification:

```
java

class WhatsAppNotification implements Notification {
    public void notifyUser(String message) {
        System.out.println("WhatsApp: " + message);
    }
}

class WhatsAppNotificationFactory implements NotificationFactory {
    public Notification createNotification() {
        return new WhatsAppNotification();
    }
}
```

👉 Just register it at startup:

```
java

FactoryRegistry.register("WHATSAPP", new WhatsAppNotificationFactory());
```

🔥 No existing code changes.
🔥 Client stays the same.
🔥 OCP respected ✅.

✅ Solution 2: Dependency Injection (Spring / Guice)

In frameworks like **Spring**, you don't even write the registry.

- You annotate classes with `@Component`, `@Bean`.
- Spring automatically wires them by type/qualifier.
- Adding WhatsAppNotification = add a class → no change to existing code.

📝 Key Points

- Your concern about **modification = violation** is 100% correct.
- The way around it is **dynamic registration** or **IoC containers**.
- Factory Method by itself solves **decoupling**, but for full **OCP compliance**, we need a registry or DI framework.

Would you like me to **take Level 6** where I show how **Spring Boot handles this problem internally** with DI + Factory Method (like how it picks database drivers or notification beans automatically)?

why we using abstract class instead of interface

Great catch Divakar 👍 — this is a very common confusion in **Factory Method pattern**.
Let's break it step by step 🎯

🎯 Why Abstract Class instead of Interface in Factory Method?

1️⃣ Factory Method Structure

Factory Method has **two main roles**:

- **Creator (Factory Class)** → defines the factory method.
- **Product (Object created)** → interface/abstract class implemented by concrete products.

Example:

```
java

abstract class Logistics {              // Creator
    public abstract Transport createTransport();  // Factory Method
    public void planDelivery() {        // Common business logic
        Transport t = createTransport();
        t.deliver();
    }
}
```

2️⃣ Why not just an interface for Creator?

If we made **`Logistics`** an **interface**:

```
java

interface Logistics {
    Transport createTransport();
    void planDelivery();  // must be implemented by all
}
```

👉 Problem: Every subclass (**`RoadLogistics`**, **`SeaLogistics`**) would have to **duplicate** the **`planDelivery()`** logic.

- That defeats the DRY (Don't Repeat Yourself) principle.
- Abstract class allows **shared code** for business logic, while still letting subclasses define the factory method.

3️⃣ Key Difference

- **Abstract Class**
  - Can provide **partial implementation** (shared methods).
  - Subclasses only override **what's different** (factory method).
  - Best when you want **common template + variation points**.
- **Interface**
  - Only contracts (no implementation, unless default methods in Java 8+).
  - Every subclass must **reimplement everything**, even common logic.
  - Leads to duplication in Factory Method pattern.

4️⃣ Example with Abstract Class ✅

```
java

abstract class NotificationFactory {
    // Factory Method
```

```
    public abstract Notification createNotification();

    // Common business logic (no need to repeat in every subclass)
    public void send(String message) {
        Notification n = createNotification();
        n.notifyUser(message);
    }
}

class EmailNotificationFactory extends NotificationFactory {
    public Notification createNotification() {
        return new EmailNotification();
    }
}
```

👉 Here, `send()` is reused by all factories.

5️⃣ Example with Interface ❌ (More Boilerplate)

```
java

interface NotificationFactory {
    Notification createNotification();
    void send(String message); // every class must reimplement
}

class EmailNotificationFactory implements NotificationFactory {
    public Notification createNotification() {
        return new EmailNotification();
    }
    public void send(String message) {   // DUPLICATE CODE
        createNotification().notifyUser(message);
    }
}
```

👉 Every factory (`Email`, `SMS`, `Push`) must duplicate the `send()` method.

6️⃣ Real-World Analogy

- **Abstract Class** → A "template" with some default behavior (like a skeleton plan). Example: "Logistics Company has a default process, only transport type varies."
- **Interface** → Just a contract: "You must provide everything yourself." No shared logic.

✅ Conclusion

We use **abstract class in Factory Method** because:

- It allows **common reusable business logic** (e.g., `send()`, `planDelivery()`).
- Subclasses only **override factory method** (the part that varies).
- It avoids **duplication** and keeps code **cleaner**.

👉 If there's no common logic at all, an **interface** could also work. But usually in Factory Method, we want a **template with shared flow** → abstract class is the best fit.

Would you like me to show you a **Level 7 example** where we use **Template Method + Factory Method** together (the real reason abstract class shines)?

```
package FactoryMethod;

interface Food{
    void cook();
}

class Pizza implements Food{
    public void cook(){
        System.out.println("Pizza Ready...");
    }
}
class Briyani implements Food{
    public void cook(){
        System.out.println("Briyani ready...");
    }
}

abstract class FoodFactory{
    abstract Food createFood();
    public void delivery(){
        Food f=createFood();
        f.cook();
    }
}

class PizzaFoodFactory extends FoodFactory{
    public Food createFood(){
        return new Pizza();
    }
}

class BriyaniFoodFactory extends FoodFactory{
    public void createFood(){
        return new Briyani();
    }
}


public class FoodFactory {
    public static void main(String args[]){
        PizzaFoodFactory pizza=new PizzaFoodFactory();
        pizza.createFood();

        BriyaniFoodFactory briyani=new BriyaniFoodFactory();
        briyani.createFood();
```

```
    }
  }
```

Nice attempt Divakar 👋, you're on the right path.
But I see **two main issues** in your code that are causing confusion/errors:

❌ Problems in Your Code

1. Method signature mismatch in `BriyaniFoodFactory`

```
java

class BriyaniFoodFactory extends FoodFactory{
    public void createFood(){   // ❌ Wrong return type
        return new Briyani();
    }
}
```

👉 The parent method is declared as:

```
java

abstract Food createFood();
```

So the overriding method **must return `Food`**, not **`void`**.

Correct version:

```
java

class BriyaniFoodFactory extends FoodFactory {
    public Food createFood() {
        return new Briyani();
    }
}
```

---

2. Wrong main class declaration

```
java

public class FoodFactory {
    ...
}
```

❌ This collides with your abstract **`FoodFactory`** class.
👉 You cannot have **two classes with the same name in one package**.

Fix → Rename your **`main`** class:

```
java

public class FoodFactoryDemo {
    public static void main(String args[]){
        FoodFactory pizzaFactory = new PizzaFoodFactory();
```

```
        pizzaFactory.delivery();

        FoodFactory briyaniFactory = new BriyaniFoodFactory();
        briyaniFactory.delivery();
    }
}
```

---

3. You are calling `createFood()` directly

In the **Factory Method pattern**, the **client should call `delivery()`**, not `createFood()`.

- `createFood()` is a factory method (decides product).
- `delivery()` is the **business method** that uses the factory method internally.

So instead of:

```
java

pizza.createFood();
briyani.createFood();
```

You should call:

```
java

pizzaFactory.delivery();
briyaniFactory.delivery();
```

✅ Corrected Full Code

```
java

package FactoryMethod;

// Product interface
interface Food {
    void cook();
}

// Concrete Products
class Pizza implements Food {
    public void cook() {
        System.out.println("Pizza Ready...");
    }
}

class Briyani implements Food {
    public void cook() {
        System.out.println("Briyani Ready...");
    }
}

// Creator (abstract factory class)
abstract class FoodFactory {
    abstract Food createFood();  // Factory Method

    // Common business logic
    public void delivery() {
        Food f = createFood();
        f.cook();
```

```
    }
}

// Concrete Factories
class PizzaFoodFactory extends FoodFactory {
    public Food createFood() {
        return new Pizza();
    }
}

class BriyaniFoodFactory extends FoodFactory {
    public Food createFood() {
        return new Briyani();
    }
}

// Client
public class FoodFactoryDemo {
    public static void main(String args[]) {
        FoodFactory pizzaFactory = new PizzaFoodFactory();
        pizzaFactory.delivery();   // Output: Pizza Ready...

        FoodFactory briyaniFactory = new BriyaniFoodFactory();
        briyaniFactory.delivery(); // Output: Briyani Ready...
    }
}
```

✅ Output

```
nginx


Pizza Ready...
Briyani Ready...
```

🔑 Key Takeaways

1. **Abstract class** allows common logic (`delivery()`) → no duplication.
2. Always **override method signature exactly** (`Food createFood()`).
3. Don't duplicate class names (`FoodFactory` vs `FoodFactoryDemo`).
4. Client should call **business method (`delivery()`)**, not the factory method directly.

👉 Do you want me to extend this example into a **real-world Restaurant System** (Level 7), where the Factory decides food dynamically based on input (like `"PIZZA"` or `"BRIYANI"`), so client doesn't need to know which factory class to call?

```
package FactoryMethod;

interface Food{
   void cook();
}

class Pizza implements Food{
```

```
    public void cook(){
        System.out.println("Pizza Ready...");
    }
}
class Briyani implements Food{
    public void cook(){
        System.out.println("Briyani ready...");
    }
}

abstract class FoodFactory{
    abstract Food createFood();
    public void serve(){
        Food f=createFood();
        f.cook();
    }
}

class PizzaFoodFactory extends FoodFactory{
    public Food createFood(){
        return new Pizza();
    }
}

class BriyaniFoodFactory extends FoodFactory{
    public Food createFood(){
        return new Briyani();
    }
}


public class FoodFactoryDemo {
    public static void main(String args[]){
        FoodFactory pizza=new PizzaFoodFactory();
        pizza.serve();
        FoodFactory briyani=new BriyaniFoodFactory();
        briyani.serve();

    }
}


does this is 100% factory method
```

Excellent question Divakar 👍
Let's analyze carefully whether your code is a **100% correct Factory Method pattern implementation**.

✅ What the Factory Method Pattern Requires

The **Factory Method** pattern says:

1. **Define a common Product interface** → (`Food`).
2. **Provide multiple Concrete Products** → (`Pizza`, `Briyani`).
3. **Define an abstract Creator class** that declares a factory method (`createFood()`).
4. **Concrete Creator classes** override the factory method and return specific product (`PizzaFoodFactory`, `BriyaniFoodFactory`).
5. The **client interacts with Creator**, not with concrete products directly.
    - Client calls business logic (`serve()`), which internally uses the factory method.

🔎 Your Implementation vs Pattern

- **Product interface** ✅ → `Food`
- **Concrete Products** ✅ → `Pizza`, `Briyani`
- **Abstract Creator with Factory Method** ✅ → `FoodFactory.createFood()`
- **Concrete Creators** ✅ → `PizzaFoodFactory`, `BriyaniFoodFactory`
- **Client calls Creator method, not directly instantiating Product** ✅ → `pizza.serve()`, `briyani.serve()`

👉 Yes, this is a **100% correct Factory Method implementation** 🎉

📝 Minor Improvements

Right now, your client still **chooses which factory** to create (`new PizzaFoodFactory()`, `new BriyaniFoodFactory()`).

That is acceptable in Factory Method, but sometimes people mix it with **Simple Factory** or **Abstract Factory** to remove that responsibility.

For pure **Factory Method**:

- Client decides **which factory to use**,
- Each factory decides **which product to create**.

So ✅ your implementation is fully Factory Method.

⚡ Real-World Analogy

- `FoodFactory` = Kitchen
- `PizzaFoodFactory` = Italian Kitchen
- `BriyaniFoodFactory` = Indian Kitchen
- `createFood()` = Cook food in style of kitchen

- `serve()` = Deliver food

The **customer (client)** chooses which kitchen to order from, but does not worry about how food is cooked.

✅ Conclusion:
Yes Divakar, your code is **100% Factory Method Pattern**.
Only note: if you want to remove the need for `new PizzaFoodFactory()`, that moves toward **Simple Factory** or **Abstract Factory**.

Do you want me to show you a **comparison between Factory Method vs Simple Factory vs Abstract Factory** with the same Food example, so you can clearly see where your current implementation fits?

next

Perfect 🚀 Divakar, let's **Level Up** our Factory Method learning by comparing it with its "cousins":

👉 **Simple Factory vs Factory Method vs Abstract Factory**

We'll keep the same **Food Example** so you can directly see the difference.

🍽️ LEVEL NEXT – Comparing Factory Families

1️⃣ Simple Factory (Not a Design Pattern, but a simple encapsulation)

- One class decides **what product to create** based on input.
- Client doesn't worry about `new`.

```
java

// Product
interface Food { void cook(); }

class Pizza implements Food {
    public void cook() { System.out.println("Pizza Ready..."); }
}
class Briyani implements Food {
    public void cook() { System.out.println("Briyani Ready..."); }
}

// Simple Factory
class SimpleFoodFactory {
    public static Food createFood(String type) {
        if(type.equalsIgnoreCase("pizza")) return new Pizza();
        else if(type.equalsIgnoreCase("briyani")) return new Briyani();
        throw new IllegalArgumentException("Unknown Food: " + type);
    }
}

// Client
```

```
public class SimpleFactoryDemo {
    public static void main(String[] args) {
        Food f1 = SimpleFoodFactory.createFood("pizza");
        f1.cook();

        Food f2 = SimpleFoodFactory.createFood("briyani");
        f2.cook();
    }
}
```

✅ Pros:

- Centralized creation, easy to use.

❌ Cons:

- Violates **Open-Closed Principle** → every time you add a new food, you must modify `SimpleFoodFactory`.

2️⃣ Factory Method (Your implementation ✅)

- Abstract creator defines a **factory method**.
- Subclasses decide which product to create.

```
java

// Product
interface Food { void cook(); }

class Pizza implements Food { public void cook(){ System.out.println("Pizza Ready..."); } }
class Briyani implements Food { public void cook(){ System.out.println("Briyani Ready..."); } }

// Abstract Creator
abstract class FoodFactory {
    abstract Food createFood();
    public void serve() {
        Food f = createFood();
        f.cook();
    }
}

// Concrete Creators
class PizzaFoodFactory extends FoodFactory {
    public Food createFood() { return new Pizza(); }
}
class BriyaniFoodFactory extends FoodFactory {
    public Food createFood() { return new Briyani(); }
}

// Client
public class FactoryMethodDemo {
    public static void main(String[] args) {
        FoodFactory pizzaFactory = new PizzaFoodFactory();
        pizzaFactory.serve();

        FoodFactory briyaniFactory = new BriyaniFoodFactory();
        briyaniFactory.serve();
    }
}
```

✅ Pros:

- Follows **Open-Closed Principle**.
- Adding new food = new factory, no modification needed.

❌ Cons:

- Still client must know **which factory** to use.

3️⃣ Abstract Factory (Factory of Factories)

- Used when you need to create **families of related products**.
- Example: IndianKitchen (Briyani + Lassi) vs ItalianKitchen (Pizza + Pasta).

```
java


// Products
interface MainCourse { void cook(); }
interface Drink { void prepare(); }

class Briyani implements MainCourse { public void cook(){ System.out.println("Briyani
Ready..."); } }
class Lassi implements Drink { public void prepare(){ System.out.println("Lassi Ready..."); } }

class Pizza implements MainCourse { public void cook(){ System.out.println("Pizza Ready..."); }
}
class Wine implements Drink { public void prepare(){ System.out.println("Wine Ready..."); } }

// Abstract Factory
interface KitchenFactory {
    MainCourse createMainCourse();
    Drink createDrink();
}

// Concrete Factories
class IndianKitchen implements KitchenFactory {
    public MainCourse createMainCourse() { return new Briyani(); }
    public Drink createDrink() { return new Lassi(); }
}

class ItalianKitchen implements KitchenFactory {
    public MainCourse createMainCourse() { return new Pizza(); }
    public Drink createDrink() { return new Wine(); }
}

// Client
public class AbstractFactoryDemo {
    public static void main(String[] args) {
        KitchenFactory indianKitchen = new IndianKitchen();
        indianKitchen.createMainCourse().cook();
        indianKitchen.createDrink().prepare();

        KitchenFactory italianKitchen = new ItalianKitchen();
        italianKitchen.createMainCourse().cook();
        italianKitchen.createDrink().prepare();
    }
}
```

✅ Pros:

- Perfect for **related products** (e.g., MainCourse + Drink).
- Families of products stay consistent.

❌ Cons:

- More complex.

⚡ Quick Comparison Table

Pattern	Who Decides Creation?	Open-Closed Principle?	Use Case
Simple Factory	One class, central `if-else`	❌ Broken (modify code for new product)	Small apps, quick solution
Factory Method	Subclass factories	✅ Followed	When you want to defer creation to subclasses
Abstract Factory	Factory of factories	✅ Followed	When you need related products (families)

🔥 So Divakar, your last code is **Factory Method 100% correct**.
The **next level** understanding is → know **when to use Simple Factory vs Factory Method vs Abstract Factory**.

👉 Want me to now show you **a real-world example** (like GUI framework buttons: WindowsButton, MacButton, LinuxButton) to see how Factory Method shines in practice?

explain my code , how to extend

Perfect Divakar, let's analyze your **FoodFactoryDemo** code carefully and see **how to extend it** in a real-world scalable way.

🔎 Your Current Code

```java

interface Food {
    void cook();
}

class Pizza implements Food {
    public void cook() { System.out.println("Pizza Ready..."); }
}

class Briyani implements Food {
    public void cook() { System.out.println("Briyani Ready..."); }
}

abstract class FoodFactory {
    abstract Food createFood();
    public void serve() {
        Food f = createFood();
        f.cook();
    }
}
```

```
class PizzaFoodFactory extends FoodFactory {
    public Food createFood() { return new Pizza(); }
}

class BriyaniFoodFactory extends FoodFactory {
    public Food createFood() { return new Briyani(); }
}

public class FoodFactoryDemo {
    public static void main(String args[]) {
        FoodFactory pizza = new PizzaFoodFactory();
        pizza.serve();

        FoodFactory briyani = new BriyaniFoodFactory();
        briyani.serve();
    }
}
```

✅ How It Works (Step by Step)

1. **Product Interface** → `Food`
   - Defines a **common contract**: `cook()`.
   - Any food (Pizza, Briyani) must implement this.
2. **Concrete Products** → `Pizza` and `Briyani`
   - Implements `cook()` with its own behavior.
3. **Abstract Creator** → `FoodFactory`
   - Declares the **Factory Method**: `createFood()` (abstract).
   - Defines **business logic** in `serve()`:

     ```
     java

     Food f = createFood();
     f.cook();
     ```

     So **client calls serve()**, not `createFood()` directly.
4. **Concrete Creators** → `PizzaFoodFactory` and `BriyaniFoodFactory`
   - Overrides `createFood()` to return specific product.
   - Encapsulates **product creation logic**.
5. **Client** → `FoodFactoryDemo`
   - Chooses which factory to use (`new PizzaFoodFactory()` or `new BriyaniFoodFactory()`).
   - Calls `serve()` → Factory Method decides which product to create.

🔧 How to Extend This Code

Suppose you want to add **Burger** and **Pasta**.

Step 1: Create new Product

```java
class Burger implements Food {
    public void cook() {
        System.out.println("Burger Ready...");
    }
}

class Pasta implements Food {
    public void cook() {
        System.out.println("Pasta Ready...");
    }
}
```

Step 2: Create new Concrete Factory

```java
class BurgerFoodFactory extends FoodFactory {
    public Food createFood() { return new Burger(); }
}

class PastaFoodFactory extends FoodFactory {
    public Food createFood() { return new Pasta(); }
}
```

Step 3: Use in Client

```java
FoodFactory burger = new BurgerFoodFactory();
burger.serve();

FoodFactory pasta = new PastaFoodFactory();
pasta.serve();
```

✅ Works perfectly, **no changes needed** in existing classes.

- That's **Open-Closed Principle** in action.

🔑 Tips to Make it More Scalable

1. **Dynamic Factory Selection (Optional)**
    - Instead of creating `new BurgerFoodFactory()` manually, you can use a **registry or map**:

```java
Map<String, FoodFactory> factoryMap = new HashMap<>();
factoryMap.put("PIZZA", new PizzaFoodFactory());
factoryMap.put("BRIYANI", new BriyaniFoodFactory());
factoryMap.put("BURGER", new BurgerFoodFactory());

String choice = "BURGER";
factoryMap.get(choice).serve();
```

- Now client code **doesn't need to change** for new foods.

2. **Keep Business Logic in Abstract Class**
   - Methods like `serve()` or `order()` stay in `FoodFactory`.
   - Only `createFood()` changes in subclasses.
3. **Combine with Template Method (Optional)**
   - If you have **common steps** like:
     1. Prepare ingredients
     2. Cook
     3. Pack
   - You can define `final void order()` in `FoodFactory` and let `createFood()` provide the food.

⚡ Summary

- Your code is **100% Factory Method** ✅
- To extend:
  1. Create new **Product class** (implements `Food`)
  2. Create new **Concrete Factory** (extends `FoodFactory`)
  3. Client uses the new factory → no change to existing code
- For **dynamic extensions without changing client** → use **factory registry** or **DI framework**.

If you want, I can write a **Level 6 version** of your code with **dynamic factory selection** so that **client never has to write `new PizzaFoodFactory()`**, fully OCP-compliant.

Do you want me to do that?

next

Perfect Divakar ⚡ — now let's go **Level 6** and make your Factory Method **fully dynamic and OCP-compliant**, so the **client never uses `new` directly**.

We'll implement a **Factory Registry** approach.

🎯 Level 6: Dynamic Factory Method with Registry

Step 1: Product Interface

```
java

interface Food {
    void cook();
```

```
}
```

Step 2: Concrete Products

```
java

class Pizza implements Food {
    public void cook() { System.out.println("Pizza Ready..."); }
}

class Briyani implements Food {
    public void cook() { System.out.println("Briyani Ready..."); }
}

class Burger implements Food {
    public void cook() { System.out.println("Burger Ready..."); }
}
```

Step 3: Abstract Creator

```
java

abstract class FoodFactory {
    abstract Food createFood();

    public void serve() {
        Food f = createFood();
        f.cook();
    }
}
```

Step 4: Concrete Factories

```
java

class PizzaFoodFactory extends FoodFactory {
    public Food createFood() { return new Pizza(); }
}

class BriyaniFoodFactory extends FoodFactory {
    public Food createFood() { return new Briyani(); }
}

class BurgerFoodFactory extends FoodFactory {
    public Food createFood() { return new Burger(); }
}
```

Step 5: Factory Registry

```
java

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

```
class FactoryRegistry {
    private static final Map<String, FoodFactory> registry = new HashMap<>();

    // Register factories dynamically
    public static void register(String key, FoodFactory factory) {
        registry.put(key.toUpperCase(), factory);
    }

    // Get factory without client knowing concrete class
    public static FoodFactory getFactory(String key) {
        FoodFactory factory = registry.get(key.toUpperCase());
        if(factory == null) throw new IllegalArgumentException("No factory registered for: " +
key);
        return factory;
    }
}
```

Step 6: Client (Fully Decoupled)

```
java


public class FoodFactoryDemo {
    public static void main(String[] args) {
        // Register factories (only at startup)
        FactoryRegistry.register("PIZZA", new PizzaFoodFactory());
        FactoryRegistry.register("BRIYANI", new BriyaniFoodFactory());
        FactoryRegistry.register("BURGER", new BurgerFoodFactory());

        // Client just requests by name
        FoodFactory factory = FactoryRegistry.getFactory("PIZZA");
        factory.serve();

        factory = FactoryRegistry.getFactory("BRIYANI");
        factory.serve();

        factory = FactoryRegistry.getFactory("BURGER");
        factory.serve();
    }
}
```

✅ Output

```
nginx


Pizza Ready...
Briyani Ready...
Burger Ready...
```

🔑 Advantages

1. **No client-side `new`** → client is fully decoupled.
2. **Open/Closed Principle** ✅ → add new food type without modifying existing code.
    - Just create new **`Food`** + **`FoodFactory`** and register it.
3. **Extensible & scalable** → works for 10s or 100s of products.
4. **Factory Method remains intact** → **`createFood()`** still decides the product.

⚡ How to Extend

1. Create a new product class (e.g., **`Pasta`**) implementing **`Food`**.
2. Create a new factory class (e.g., **`PastaFoodFactory`**) extending **`FoodFactory`**.
3. Register it at startup:

```
java

FactoryRegistry.register("PASTA", new PastaFoodFactory());
```

✅ Done! Client code never changes.

This is **Factory Method + OCP + Dynamic Selection** — fully production-ready.

If you want, I can make **Level 7**, where we **combine Template Method + Factory Method** to model a **full restaurant workflow** (prepare, cook, pack, deliver) showing why abstract class is essential.

Do you want me to do that?