4th

Factory Method and Abstract Factory Pattern - Detailed Explanation

Let me explain both patterns in detail with step-by-step Java implementations.

---

1. Factory Method Pattern

Definition: Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

When to use:

- When you don't know beforehand the exact types of objects your code should work with
- When you want to provide a way to extend internal components
- When you want to save system resources by reusing existing objects instead of rebuilding them

Key Components:

1. **Product** - Interface for objects the factory method creates
2. **ConcreteProduct** - Implements the Product interface
3. **Creator** - Declares the factory method
4. **ConcreteCreator** - Overrides the factory method to return ConcreteProduct

Factory Method Code Example - Step by Step

java

```
// Step 1: Create the Product interface

// This defines what all vehicles must be able to do

interface Vehicle {

    void drive();

    void specs();

}


// Step 2: Create Concrete Products
```

```
// These are the actual vehicle types we'll create
class Car implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Driving a car on the road");
    }

    @Override
    public void specs() {
        System.out.println("Car: 4 wheels, seats 5 people");
    }
}

class Bike implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Riding a bike on the road");
    }

    @Override
    public void specs() {
        System.out.println("Bike: 2 wheels, seats 2 people");
    }
```

```
}

class Truck implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Driving a truck for heavy loads");
    }

    @Override
    public void specs() {
        System.out.println("Truck: 6+ wheels, carries heavy cargo");
    }
}

// Step 3: Create the Creator (abstract class with factory method)
// This declares the factory method that subclasses must implement
abstract class VehicleFactory {

    // This is the Factory Method - subclasses will override this
    public abstract Vehicle createVehicle();

    // This is a template method that uses the factory method
    public void deliverVehicle() {
```

```
        Vehicle vehicle = createVehicle();
        System.out.println("--- Vehicle Created ---");
        vehicle.specs();
        vehicle.drive();
        System.out.println("--- Vehicle Delivered ---\n");
    }
}

// Step 4: Create Concrete Creators
// Each creator decides which product to instantiate
class CarFactory extends VehicleFactory {
    @Override
    public Vehicle createVehicle() {
        return new Car();
    }
}

class BikeFactory extends VehicleFactory {
    @Override
    public Vehicle createVehicle() {
        return new Bike();
    }
}
```

```
class TruckFactory extends VehicleFactory {
    @Override
    public Vehicle createVehicle() {
        return new Truck();
    }
}


// Step 5: Client code
public class FactoryMethodDemo {
    public static void main(String[] args) {
        // Create different factories
        VehicleFactory carFactory = new CarFactory();
        VehicleFactory bikeFactory = new BikeFactory();
        VehicleFactory truckFactory = new TruckFactory();

        // Use the factories to create and deliver vehicles
        carFactory.deliverVehicle();
        bikeFactory.deliverVehicle();
        truckFactory.deliverVehicle();
    }
}
```

How it works:

1. The client creates a specific factory (CarFactory, BikeFactory, etc.)
2. The factory's createVehicle() method returns the appropriate vehicle type
3. The deliverVehicle() method uses the factory method without knowing the concrete type
4. Each factory decides which concrete class to instantiate

---

2. Abstract Factory Pattern

Definition: Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

When to use:

- When your code needs to work with various families of related products
- When you want to ensure that products from the same family are used together
- When you want to provide a library of products and reveal only their interfaces

Key Components:

1. **AbstractProduct** - Interfaces for different product types
2. **ConcreteProduct** - Implementations of products
3. **AbstractFactory** - Interface declaring creation methods for all products
4. **ConcreteFactory** - Implements creation methods for specific product families

Abstract Factory Code Example - Step by Step

java

```
// Step 1: Create Abstract Products

// These define interfaces for different types of UI components


interface Button {

    void render();

    void onClick();

}


interface Checkbox {
```

```
    void render();
    void toggle();
}

interface TextField {
    void render();
    void setText(String text);
}

// Step 2: Create Concrete Products for Windows Family
class WindowsButton implements Button {
    @Override
    public void render() {
        System.out.println("Rendering Windows-style button with shadow");
    }

    @Override
    public void onClick() {
        System.out.println("Windows button clicked with system sound");
    }
}

class WindowsCheckbox implements Checkbox {
```

```
    @Override
    public void render() {
        System.out.println("Rendering Windows-style checkbox with blue check");
    }

    @Override
    public void toggle() {
        System.out.println("Windows checkbox toggled");
    }
}

class WindowsTextField implements TextField {
    @Override
    public void render() {
        System.out.println("Rendering Windows-style text field with border");
    }

    @Override
    public void setText(String text) {
        System.out.println("Windows TextField: " + text);
    }
}
```

```
// Step 3: Create Concrete Products for Mac Family
class MacButton implements Button {
    @Override
    public void render() {
        System.out.println("Rendering Mac-style button with gradient");
    }

    @Override
    public void onClick() {
        System.out.println("Mac button clicked with sleek animation");
    }
}

class MacCheckbox implements Checkbox {
    @Override
    public void render() {
        System.out.println("Rendering Mac-style checkbox with rounded corners");
    }

    @Override
    public void toggle() {
        System.out.println("Mac checkbox toggled smoothly");
    }
```

```
}

class MacTextField implements TextField {
    @Override
    public void render() {
        System.out.println("Rendering Mac-style text field with rounded border");
    }

    @Override
    public void setText(String text) {
        System.out.println("Mac TextField: " + text);
    }
}

// Step 4: Create Concrete Products for Linux Family
class LinuxButton implements Button {
    @Override
    public void render() {
        System.out.println("Rendering Linux-style button, flat design");
    }

    @Override
    public void onClick() {
```

```
        System.out.println("Linux button clicked");
    }
}

class LinuxCheckbox implements Checkbox {
    @Override
    public void render() {
        System.out.println("Rendering Linux-style checkbox");
    }

    @Override
    public void toggle() {
        System.out.println("Linux checkbox toggled");
    }
}

class LinuxTextField implements TextField {
    @Override
    public void render() {
        System.out.println("Rendering Linux-style text field");
    }

    @Override
```

```
    public void setText(String text) {
        System.out.println("Linux TextField: " + text);
    }
}

// Step 5: Create Abstract Factory Interface
// This declares methods for creating each type of product
interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
    TextField createTextField();
}

// Step 6: Create Concrete Factories for each family
class WindowsFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new WindowsButton();
    }

    @Override
    public Checkbox createCheckbox() {
        return new WindowsCheckbox();
```

```
    }

    @Override
    public TextField createTextField() {
        return new WindowsTextField();
    }
}

class MacFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new MacButton();
    }

    @Override
    public Checkbox createCheckbox() {
        return new MacCheckbox();
    }

    @Override
    public TextField createTextField() {
        return new MacTextField();
    }
```

```
}

class LinuxFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new LinuxButton();
    }

    @Override
    public Checkbox createCheckbox() {
        return new LinuxCheckbox();
    }

    @Override
    public TextField createTextField() {
        return new LinuxTextField();
    }
}

// Step 7: Client code that works with factories
class Application {
    private Button button;
    private Checkbox checkbox;
```

```
private TextField textField;

// Constructor receives a factory
public Application(GUIFactory factory) {
    button = factory.createButton();
    checkbox = factory.createCheckbox();
    textField = factory.createTextField();
}

public void renderUI() {
    System.out.println("=== Rendering Application UI ===");
    button.render();
    checkbox.render();
    textField.render();
    System.out.println();
}

public void interactWithUI() {
    System.out.println("=== User Interactions ===");
    button.onClick();
    checkbox.toggle();
    textField.setText("Hello, World!");
    System.out.println();
```

```
    }
}

// Step 8: Demo class
public class AbstractFactoryDemo {
    public static void main(String[] args) {
        // Determine OS (in real app, this would be detected)
        String osType = "Windows"; // Could be "Windows", "Mac", or "Linux"

        GUIFactory factory;

        // Choose factory based on OS
        if (osType.equals("Windows")) {
            factory = new WindowsFactory();
        } else if (osType.equals("Mac")) {
            factory = new MacFactory();
        } else {
            factory = new LinuxFactory();
        }

        // Create application with the chosen factory
        Application app = new Application(factory);
        app.renderUI();
```

```
        app.interactWithUI();


        // Test with different OS
        System.out.println("\n--- Switching to Mac ---\n");
        Application macApp = new Application(new MacFactory());
        macApp.renderUI();
        macApp.interactWithUI();
    }
}
```

How it works:

1. The client determines which factory to use (WindowsFactory, MacFactory, etc.)
2. The factory is passed to the Application
3. The Application uses the factory to create all UI components
4. All components belong to the same family (all Windows or all Mac)
5. The Application code doesn't need to know the concrete classes

---

Key Differences Between Factory Method and Abstract Factory

Aspect	Factory Method	Abstract Factory
Purpose	Creates ONE type of product	Creates FAMILIES of related products
Complexity	Simpler, single product creation	More complex, multiple product types
Structure	Uses inheritance (subclasses decide)	Uses composition (factory objects)
Product Variety	Different variations of one product	Multiple related products together
Example	CarFactory creates Car, BikeFactory creates Bike	WindowsFactory creates Button+Checkbox+TextField for Windows

When to choose which:

- Use **Factory Method** when you have a single product hierarchy and subclasses need to specify which product to create
- Use **Abstract Factory** when you need to create families of related products that must be used together