Mastering Factory Patterns: Simple Factory, Factory Method, and Abstract Factory

The Factory patterns are key design paradigms in software development. They provide flexible object creation mechanisms, improve code maintainability, and adhere to key object-oriented principles. Let's explore **Simple Factory**, **Factory Method**, and **Abstract Factory** with code examples and detailed explanations.

---

1. Simple Factory

The Simple Factory isn't a formal design pattern but a commonly used technique to encapsulate object creation. It centralizes the instantiation logic, reducing duplication and improving code clarity.

How It Works

1. Define a **base product class or interface** (e.g., `Pizza`).
2. Create **concrete product classes** (e.g., `CheesePizza`, `PepperoniPizza`).
3. Implement a **factory class** (e.g., `SimplePizzaFactory`) that handles object creation.
4. The client calls the factory instead of directly instantiating objects.

Steps to Implement

- **Step 1**: Define a base class or interface.
- **Step 2**: Create multiple concrete implementations of the base class.
- **Step 3**: Write a factory class with a method to create and return objects based on input.
- **Step 4**: Use the factory class in your client code.

Code

```java
// Base Product
public abstract class Pizza {
    public abstract String getDescription();
    public void prepare() {
        System.out.println("Preparing " + getDescription());
    }
}
```

```java
// Concrete Products
public class CheesePizza extends Pizza {
    public String getDescription() {
        return "Cheese Pizza";
    }
}

public class PepperoniPizza extends Pizza {
    public String getDescription() {
        return "Pepperoni Pizza";
    }
}

// Factory Class
public class SimplePizzaFactory {
    public Pizza createPizza(String type) {
        return switch (type.toLowerCase()) {
            case "cheese" -> new CheesePizza();
            case "pepperoni" -> new PepperoniPizza();
            default -> throw new
        IllegalArgumentException("Unknown pizza type: " + type);
        };
    }
}

// Client Code
public class PizzaStoreSimulator {
    public static void main(String[] args) {
        SimplePizzaFactory factory = new SimplePizzaFactory();
        Pizza pizza = factory.createPizza("cheese");
        pizza.prepare();
    }
}
```

---

2. Factory Method

The Factory Method is a formal design pattern that provides a way to delegate the instantiation logic to subclasses. Each subclass defines its own way of creating specific objects.

How It Works

1. Define an **abstract creator class** (e.g., `PizzaStore`) with a factory method (e.g., `createPizza`).

2. Subclasses (e.g., `NYPizzaStore`, `ChicagoPizzaStore`) override the factory method to create specific products.
3. The client interacts with the creator class, which delegates object creation to its subclasses.

Steps to Implement

- **Step 1**: Define an abstract creator with a factory method.
- **Step 2**: Create concrete implementations of the creator class.
- **Step 3**: Override the factory method to produce specific products.
- **Step 4**: Use the creator class in your client code.

Code

```java
// Abstract Creator
public abstract class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);
        pizza.prepare();
        return pizza;
    }

    protected abstract Pizza createPizza(String type);
}

// Concrete Creators
public class NYPizzaStore extends PizzaStore {
    protected Pizza createPizza(String type) {
        if (type.equals("cheese")) {
            return new CheesePizza();
        } else if (type.equals("pepperoni")) {
            return new PepperoniPizza();
        }
        throw new IllegalArgumentException("Unknown pizza type: "
        + type);
    }
}

public class ChicagoPizzaStore extends PizzaStore {
    protected Pizza createPizza(String type) {
        if (type.equals("cheese")) {
            return new CheesePizza();
        } else if (type.equals("pepperoni")) {
            return new PepperoniPizza();
        }
        throw new IllegalArgumentException("Unknown pizza type: "
        + type);
    }
}

// Client Code
public class FactoryMethodSimulator {
    public static void main(String[] args) {
        PizzaStore nyStore = new NYPizzaStore();
        nyStore.orderPizza("cheese");

        PizzaStore chicagoStore = new ChicagoPizzaStore();
        chicagoStore.orderPizza("pepperoni");
    }
}
```

---

3. Abstract Factory

The Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. It’s useful when you need to ensure compatibility between related products.

How It Works

1. Define an **abstract factory interface** (e.g., `IngredientFactory`) for creating families of products.
2. Implement **concrete factories** (e.g., `NYIngredientFactory`, `ChicagoIngredientFactory`) for specific product families.
3. Use these factories in your main application to produce compatible objects.

Steps to Implement

- **Step 1**: Define a base abstract factory interface.
- **Step 2**: Implement concrete factories for specific families.
- **Step 3**: Use the factories in your business logic to create objects.

Code

```java
// Abstract Factory
public interface IngredientFactory {
    String createDough();
    String createSauce();
    String createCheese();
}
```

```java
// Concrete Factories
public class NYIngredientFactory implements IngredientFactory {
    public String createDough() {
        return "Thin Crust Dough";
    }
    public String createSauce() {
        return "Marinara Sauce";
    }
    public String createCheese() {
        return "Reggiano Cheese";
    }
}

public class ChicagoIngredientFactory implements
        IngredientFactory {
    public String createDough() {
        return "Thick Crust Dough";
    }
    public String createSauce() {
        return "Plum Tomato Sauce";
    }
    public String createCheese() {
        return "Mozzarella Cheese";
    }
}

// Client Usage
public class AbstractFactorySimulator {
    public static void main(String[] args) {
        IngredientFactory nyFactory = new NYIngredientFactory();
        System.out.println("NY Ingredients: " +
        nyFactory.createDough() + ", " +
        nyFactory.createSauce());

        IngredientFactory chicagoFactory = new
        ChicagoIngredientFactory();
        System.out.println("Chicago Ingredients: " +
        chicagoFactory.createDough() + ", " +
        chicagoFactory.createSauce());
    }
}
```

Summary

Pattern	Key Feature	Example
Simple Factory	Encapsulates object creation logic	Centralized pizza creation
Factory Method	Delegates creation to subclasses	Region-specific pizza stores
Abstract Factory	Produces families of related objects	Ingredient factories for NY/Chicago

Explore these patterns in your projects to achieve scalable and maintainable designs!