

# Factory Pattern

---

514770-1  
Fall 2021  
10/4/2021  
Kyoung Shin Park  
Computer Engineering  
Dankook University

## Factory Method Pattern

---

- "Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses."
- Also known as "**Virtual Constructor**".
- **The "new" operator considered harmful.**
- Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
- Factory pattern is one of the most used design pattern in Java.

## Factory Method Pattern

---

- `java.util.Calendar#getInstance()`
- `java.util.ResourceBundle#getBundle()`
- `java.text.NumberFormat#getInstance()`
- `java.nio.charset.Charset#forName()`
- `java.net.URLStreamHandlerFactory#createURLStreamHandler(String)`
- `java.util.EnumSet#of()`
- `javax.xml.bind.JAXBContext#createMarshaller()`

## Abstract Factory Pattern

---

- "Provide an interface for creating families of related or dependent objects without specifying their concrete classes."
- A hierarchy that encapsulates many possible "platforms", and the construction of a suite of "products"
- Also known as "**Factory of Factories**"
- The "new" operator considered harmful.
- Lets you produce families of related objects without specifying their concrete classes.

## Abstract Factory Pattern

- ❑ `javax.xml.parsers.DocumentBuilderFactory#newInstance()`
- ❑ `javax.xml.transform.TransformerFactory#newInstance()`
- ❑ `javax.xml.xpath.XPathFactory#newInstance()`

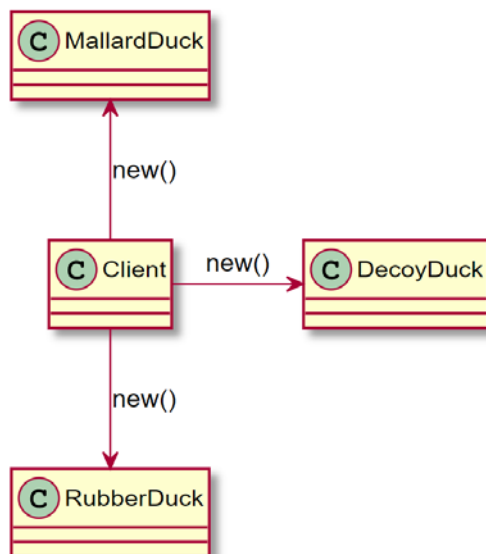
## Problem

- ❑ Problem with "new"
  - "new" instantiates a concrete class, so that's definitely an implementation, not an interface.
  - This example shows different duck classes, and we don't know until runtime which one we need to instantiate.

```
Duck duck;
if (picnic) {
    duck = new MallardDuck();
} else if (hunting) {
    duck = new DecoyDuck();
} else if (inBathTub) {
    duck = new RubberDuck();
}
```

- OCP violation (not closed for modification)
  - ❑ Code needs to be modified when it's time for change or extension
  - ❑ Making maintenance and updates more difficult and error-prone

## Problem



## Factory Pattern

	Description
Pattern	Factory Method, Abstract Factory
Problem	Whenever creating an object using <code>new()</code> , it violates <b>principle of programming for interface rather than implementation</b> which eventually result in inflexible code and difficult to change in maintenance. Another problem is class needs to contain objects of other classes or class hierarchies within it; this can be very easily achieved by just using <code>new()</code> . This is a very hard coded approach to create objects as this creates dependency between the two classes.
Solution	All factories encapsulate object creation.
Result	Factory Pattern promotes <b>loose coupling</b> by eliminating the need to bind application-specific classes into the code. Dependency Inversion Principle

## Pizza Store (HFDP Ch. 4)

- Let's say you have a pizza shop in Objectville.
- You might end up writing some code like this..

```
void prepareToBoxing(Pizza pizza) {
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
}

Pizza orderPizza() {
    Pizza pizza = new Pizza();

    prepareToBoxing(pizza);
    return pizza;
}
```

## Pizza Store (HFDP Ch. 4)

- But you need *more than one type of pizza*

```
Pizza orderPizza(String type) {
    Pizza pizza;

    if (type.equals("cheese")) {
        pizza = new CheesePizza();
    } else if (type.equals("greek")) {
        pizza = new GreekPizza();
    } else if (type.equals("pepperoni")) {
        pizza = new PepperoniPizza();
    }

    prepareToBoxing(pizza);
    return pizza;
}
```

Instantiate the correct concrete class based on the type of pizza

## Pizza Store (HFDP Ch. 4)

- **This code is NOT closed for modification.**

```
Pizza orderPizza(String type) {
    Pizza pizza;
    if (type.equals("cheese")) {
        pizza = new CheesePizza();
    } else if (type.equals("greek")) {
        pizza = new GreekPizza();
    } else if (type.equals("pepperoni")) {
        pizza = new PepperoniPizza();
    } else if (type.equals("clam")) {
        pizza = new ClamPizza();
    } else if (type.equals("veggie")) {
        pizza = new VeggiePizza();
    }
    prepareToBoxing(pizza);
    return pizza;
}
```

This is what varies.

This is what we expect to stay the same.

## Pizza Store (HFDP Ch. 4)

- **Encapsulating object creation**

```
public class SimplePizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza = null;
        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

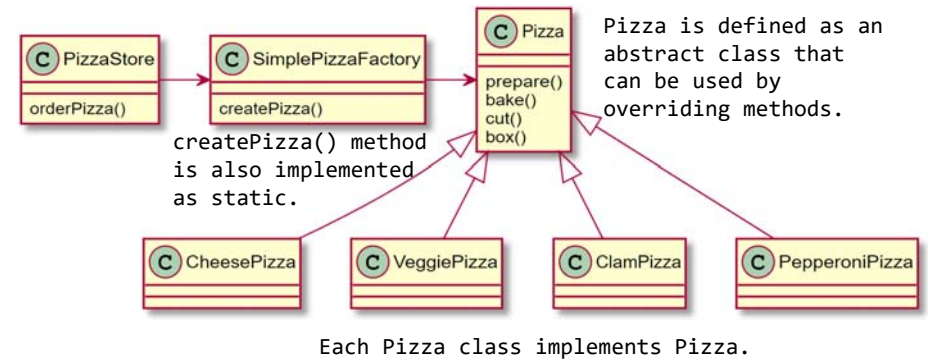
## Pizza Store (HFDP Ch. 4)

- Building a SimplePizzaFactory and reworking the PizzaStore class

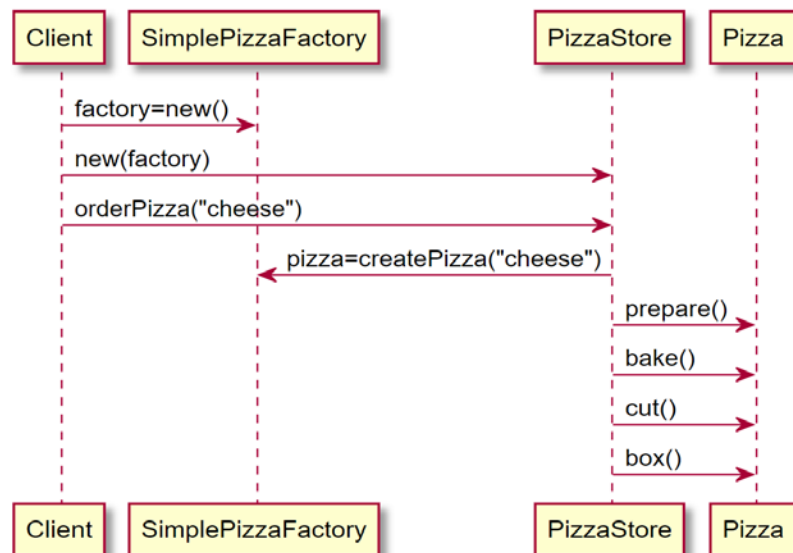
```
public class PizzaStore {
    SimplePizzaFactory factory;
    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }
    public Pizza orderPizza(String type) {
        Pizza pizza = null;
        pizza = factory.createPizza(type);
        prepareToBoxing(pizza);
        return pizza;
    }
    void prepareToBoxing(Pizza pizza) {
        ... // 기존 코드
    }
}
```

## Pizza Store (HFDP Ch. 4)

- PizzaStore Class Diagram



## Pizza Store (HFDP Ch. 4)

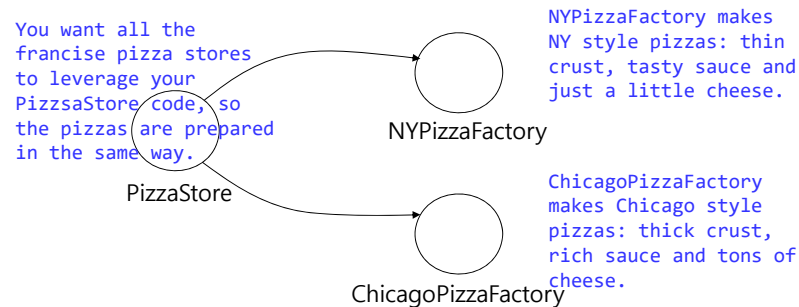


## Simple Factory

- Simple Factory determines which object to create and return the right object for user
  - In general, it determines the object to be created according to the string using the "if" statement.
- The Simple Factory isn't actually a design pattern; it's more of a programming idiom. But it is commonly used.

## Pizza Franchise (HFDP Ch. 4)

- As the franchiser, you want to ensure the quality of the franchise operations. But, each franchise might want to offer different styles of pizzas (New York, Chicago, California).



## Pizza Franchise (HFDP Ch. 4)

- If we take out SimplePizzaFactory and create 3 different factories, then we can just compose the PizzaStore with the appropriate factory.

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.orderPizza("veggie");
```

```
ChicagoPizzaFactory cFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(cFactory);
chicagoStore.orderPizza("veggie");
```

- Problem
  - Since PizzaStore is separate from the pizza creation, it guarantee the flexibility, **but it may be difficult to employ their own home grown procedures.** (orderPizza process in PizzaStore)
  - Different pizza stores may want different process.

## Pizza Franchise (HFDP Ch. 4)

- A framework that ties the **pizza store** and the **pizza creation** together, yet still allows things to **remain flexible**.
  - There is a way to localize all the pizza making activities to the PizzaStore class, and yet give the **franchises freedom to have their own regional style**.
  - Put the `createPizza()` method back into PizzaStore, but this time as an **abstract method**, and then create a PizzaStore subclass for each regional style.
  - We're going to have **a subclass for each regional type** (NYPizzaStore, ChicagoPizzaStore, CaliforniaPizzaStore) and each **subclass** is going to **make the decision** about what makes up a pizza.

## Pizza Franchise (HFDP Ch. 4)

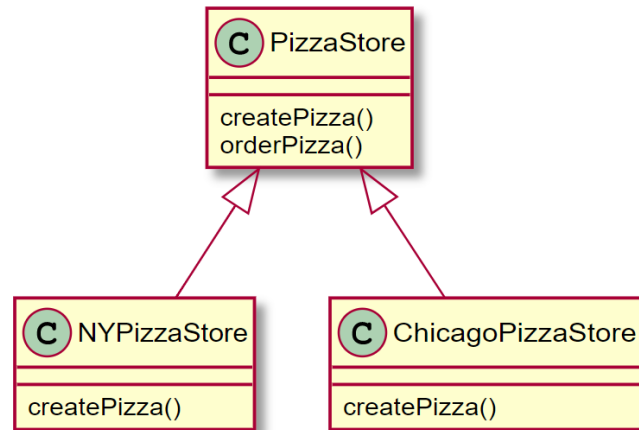
```
public abstract class PizzaStore {
    void prepareToBoxing(Pizza pizza) {
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
    }

    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);
        prepareToBoxing(pizza);
        return pizza;
    }

    // factory method
    abstract Pizza createPizza(String type);
}
```

*createPizza is back to being a call to a method in the PizzaStore rather than on a factory object.*

## Pizza Franchise (HFDP Ch. 4)



## Pizza Franchise (HFDP Ch. 4)

```
public class NYPizzaStore extends PizzaStore {
    Pizza createPizza(String type) {
        if type.equals("cheese")) {
            pizza = new NYStyleCheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new NYStylePepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new NYStyleClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new NYStyleVeggiePizza();
        }
    }
}
```

## Pizza Franchise (HFDP Ch. 4)

```
public class ChicagoPizzaStore extends PizzaStore {
    Pizza createPizza(String type) {
        if type.equals("cheese")) {
            pizza = new ChicagoStyleCheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new ChicagoStylePepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ChicagoStyleClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new ChicagoStyleVeggiePizza();
        }
    }
}
```

## Factory Method

- ❑ The **factory method** is **abstract**, so the **subclasses** are counted on to handle **object creation**.
- ❑ It can separate the client code in the superclass and the object creation code in the subclass.  
`abstract Product factoryMethod(String type)`
- ❑ The factory method returns an object of type **Product** that is typically used **within methods defined in the superclass**.
- ❑ The factory method **isolates the client** (e.g., the code in the superclass, like orderPizza()) from knowing what kind of concrete Product is actually created.

## Pizza Class

```
public abstract class Pizza {
    String name;
    String dough;
    String sauce;
    ArrayList toppings = new ArrayList();

    void prepare() {
        System.out.println("Preparing " + name);
        System.out.println("Tossing dough...");
        System.out.println("Adding sauce...");
        System.out.println("Adding toppings: ");
        for (int i = 0; i < toppings.size(); i++) {
            System.out.println(" " + toppings.get(i));
        }
    }
    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }
}
```

## Pizza Class

```
void cut() {
    System.out.println("Cutting the pizza into
diagonal slices");
}
void box() {
    System.out.println("Place pizza in official
PizzaStore box");
}
public String getName() {
    return name;
}
}
```

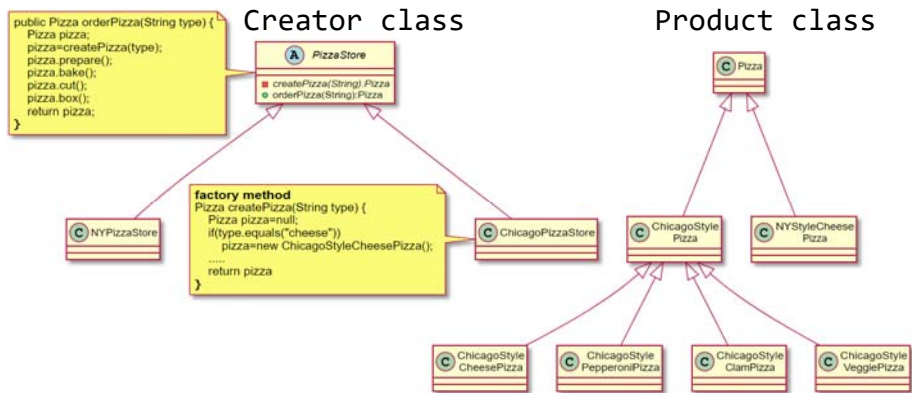
```
public class NYStyleCheesePizza extends Pizza {
    public NYStyleCheesePizza() {
        name = "NY Style Sauce and Cheese Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";
        toppings.add("Grated Reggiano Cheese");
    }
}

public class ChicagoStyleCheesePizza extends Pizza {
    public ChicagoStyleCheesePizza () {
        name = "Chicago Style Deep Dish Cheese Pizza";
        dough = "Extra Thick Crust Dough";
        sauce = "Plum Tomato Sauce";
        toppings.add("Shredded Mozzarella Cheese");
    }
    void cut() {
        System.out.println("Cutting the pizza into
square slices");
    }
}
```

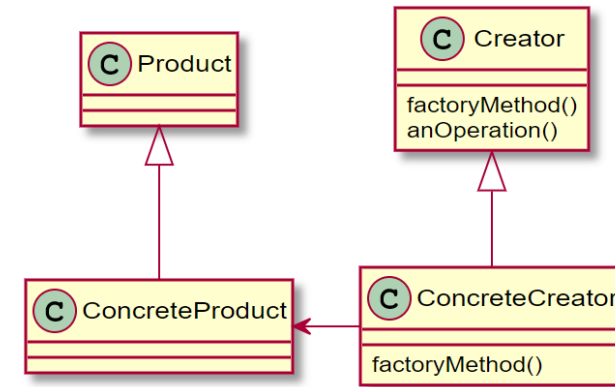
## main method

```
public class PizzaTestDrive {
    public static void main(String[] args) {
        PizzaStore nyStore = new NYPizzaStore();
        PizzaStore chicagoStore = new ChicagoPizzaStore();
        Pizza pizza = nyStore.orderPizza("cheese");
        System.out.println("Ethan ordered a "
            + pizza.getName() + "\n");
        pizza = chicagoStore.orderPizza("cheese");
        System.out.println("Joel ordered a "
            + pizza.getName() + "\n");
    }
}
```

## Factory Method Pattern



## Factory Method Pattern



## Define Factory Method Pattern

- Creator
  - Defines a method that needs to create an object whose actual type is unknown. Does so using abstract method call.
- ConcreteCreator
  - Subclass that overrides the abstract object-instantiation method to create the Concrete Product.
- Product
  - Interface implemented by the created product. Creator accesses the ConcreteProduct object through this interface.
- ConcreteProduct
  - Object used by the Creator (superclass) methods. Implements the Product interface.

## Without Factory Method Pattern?

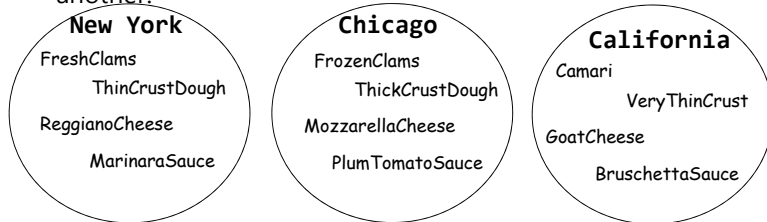
```

public class DependentPizzaStore {
    public Pizza createPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("NY")) {
            if (type.equals("cheese")) {
                pizza = new NYStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new NYStyleVeggiePizza();
            }
            ...
        }
        else if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
                pizza = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new ChicagoStyleVeggiePizza();
            }
            ...
        }
        ...
    }
}
    
```



## Families of Pizza Ingredients

- How to ensure each franchise is using **quality ingredients**?
  - You're going to build a factory that produces and ships them to your franchise.
  - The problem is that the franchise are located in different regions. New York uses one set of ingredients and Chicago another.



## Families of Pizza Ingredients

- To build the ingredient factories, let's start by defining **an interface for the factory** that is going to create all our ingredients.

```
public interface PizzaIngredientFactory {  
    public Dough createDough();  
    public Sauce createSauce();  
    public Cheese createCheese();  
    public Veggies[] createVeggies();  
    public Pepperoni createPepperoni();  
    public Clams createClam();  
}
```

## Families of Pizza Ingredients

- New York Ingredient Factory

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {  
    public Dough createDough() {  
        return new ThinCrustDough();  
    }  
    public Sauce createSauce() {  
        return new MarinaraSauce();  
    }  
    public Cheese createCheese() {  
        return new ReggianoCheese();  
    }  
    public Veggies[] createVeggies() {  
        Veggies veggies[] = { new Garlic(), new Onion(),  
                               new Mushroom(), new RedPepper() };  
        return veggies;  
    }  
}
```

## Families of Pizza Ingredients

```
    public Pepperoni createPepperoni() {  
        return new SlicedPepperoni();  
    }  
    public Clams createClam() {  
        return new FreshClams();  
    }  
}
```

## Families of Pizza Ingredients

- Write a new Pizza class

```
public abstract class Pizza {
    String name;
    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;

    abstract void prepare();

    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }
}
```

Each Pizza holds a set of ingredients that are used in its preparation.

## Families of Pizza Ingredients

```
void cut() {
    System.out.println("Cutting the pizza into diagonal slices");
}
void box() {
    System.out.println("Place pizza in official PizzaStore box");
}
void setName(String name) {
    this.name = name;
}
String getName() {
    return name;
}
public String toString() {
    // print the Pizza name
}
}
```

## Families of Pizza Ingredients

- In the factory method pattern, **NYCheesePizza** and **ChicagoCheesePizza** classes are **the same**, except that they use **regional ingredients**.
  - The pizzas are made the same (dough + sauce + cheese). They all follow the same preparation steps; they just have different ingredients.
  - So, we really don't need two classes for each pizza; **the ingredient factory is going to handle the regional differences**.

## Families of Pizza Ingredients

- CheesePizza Class

```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;
    public CheesePizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }
    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}
```

## Families of Pizza Ingredients

### ClamPizza Class

```
public class ClamPizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;
    public ClamPizza(PizzaIngredientFactory
                    ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }
    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
        clam = ingredientFactory.createClam();
    }
}
```

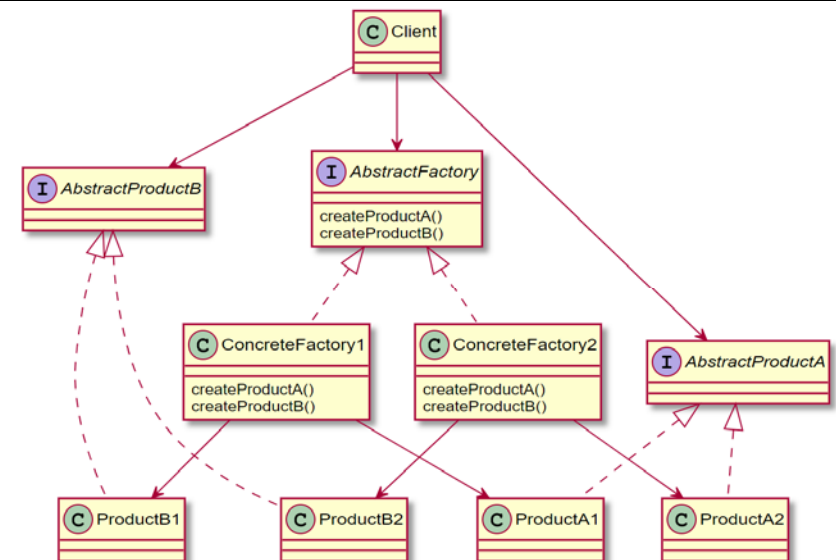
## Families of Pizza Ingredients

```
public class NYPizzaStore extends PizzaStore {
    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory =
            new NYPizzaIngredientFactory();
        if (item.equals("cheese")) {
            pizza = new CheesePizza(ingredientFactory);
            pizza.setName("New York Style Cheese Pizza");
        } else if (item.equals("veggie")) {
            pizza = new VeggiePizza(ingredientFactory);
            pizza.setName("New York Style Veggie Pizza");
        } else if (item.equals("clam")) {
            ...
        }
        return pizza;
    }
}
```

## Abstract Factory Pattern

- Abstract Factory allows a client to use **an abstract interface to create a set of related products** without knowing about the concrete products that are actually produced.
- In this way, the client is **decoupled** from any of the specifics of the concrete products.
- Abstract Factory can be used for creating cross-platform UI elements without coupling the client code to concrete UI classes, while keeping all created elements consistent with a selected operating system (Windows, Mac).
  - GUIFactory interface – createButton, createCheckBox
  - WindowsFactory – createButton creates Windows button & createCheckBox creates Windows checkbox
  - MacFactory – createButton creates Mac button & createCheckBox creates Mac checkbox

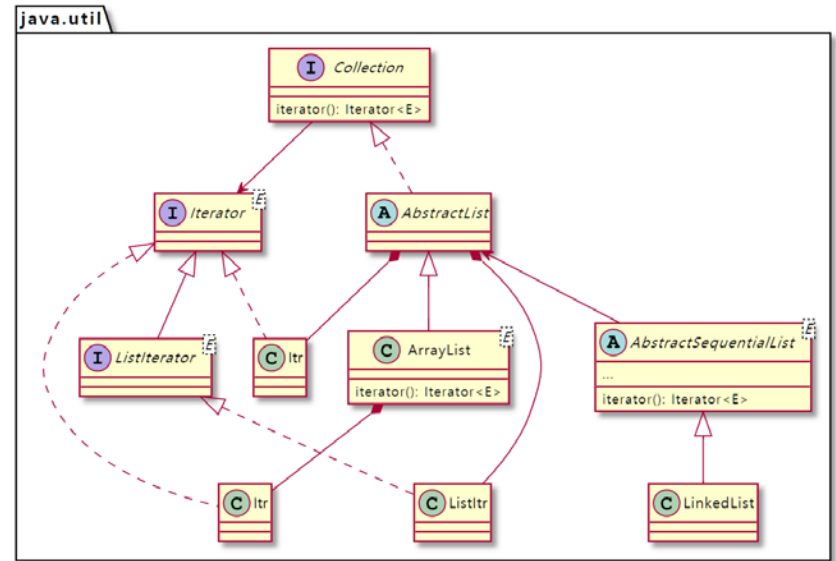
## Abstract Factory Pattern



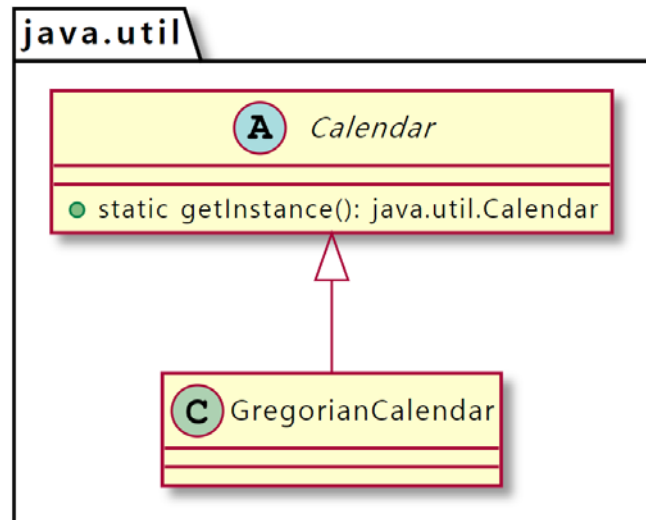
## Abstract Factory Pattern

- ❑ AbstractFactory
  - Defines the interface that all concrete factories must implement, which consists of a set of methods for creating products.
- ❑ ConcreteFactory1, ConcreteFactory2
  - Each concrete factory can produce an entire set of products.
- ❑ ProductA1, ProductA2
  - They are the product family of ProductA.
- ❑ ProductB1, ProductB2
  - They are the product family of ProductB.

## Factory Method Pattern Example



## Factory Method Pattern Example



## Difference between Abstract Factory and Factory Method

- ❑ **Abstract Factory** uses **object composition** to delegate responsibility of creating object to another class: object creation is implemented in methods exposed in the factory interface.
- ❑ **Factory Method** uses **inheritance** and relies on a subclass to create object: object creation is delegated to subclasses which implement the factory method to create objects.
- ❑ Factory Method is **just a method** that can be overridden in a subclass. Abstract Factory is **an object that has multiple factory methods** on it.