Factory Pattern
Builder Pattern

514770-1
Fall 2023
10/17/2023
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.Calender#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()

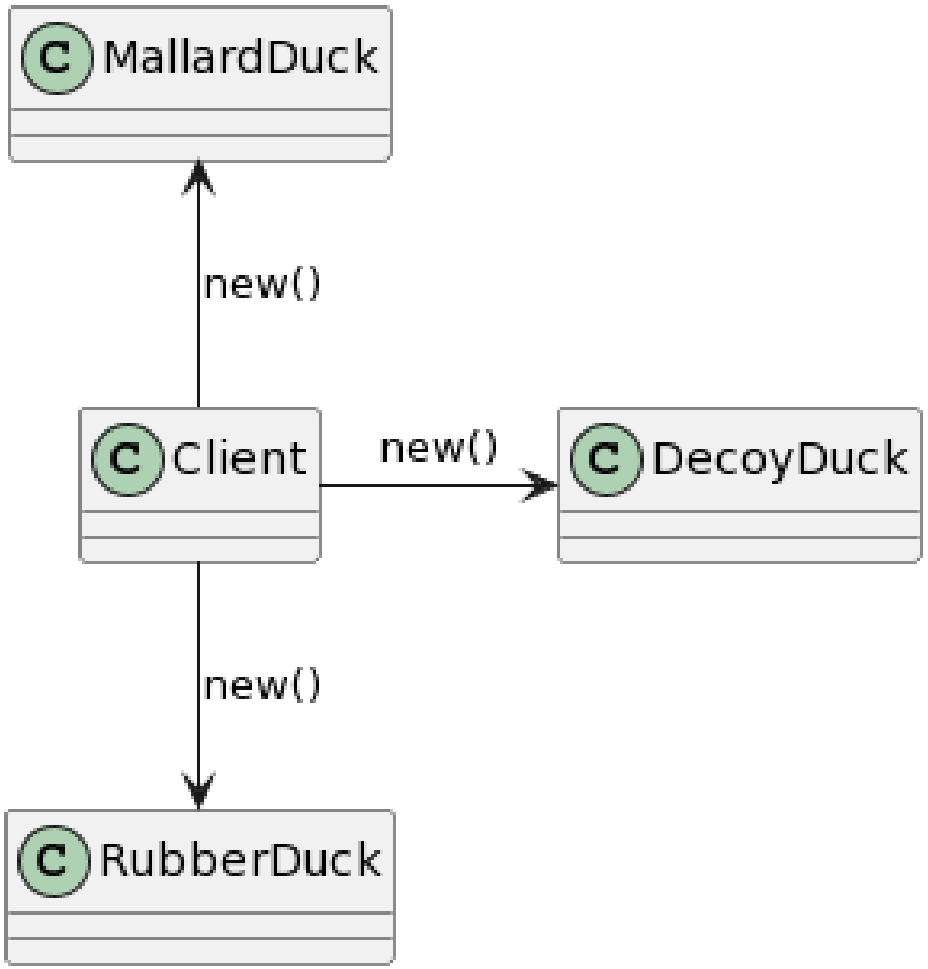
Problem

- 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 new(), it violates **principle of programming for interface rather than implementation** 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 new(). 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 **loose coupling** 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;
  }
}
```

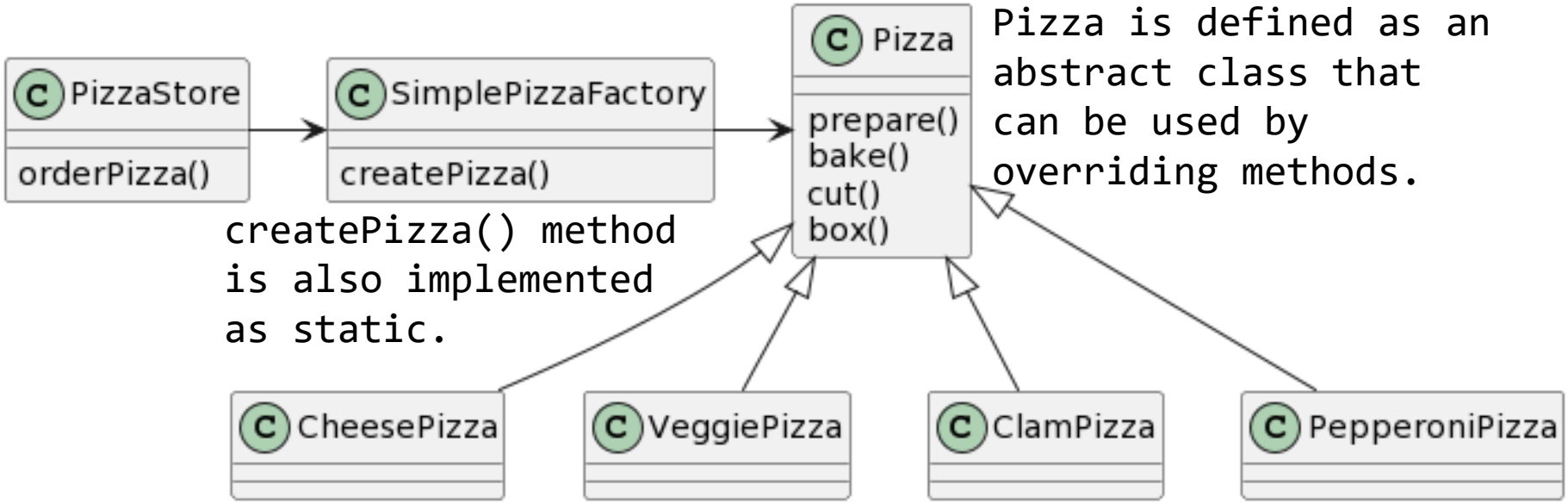
Pizza Store (HFDP Ch. 4)

- 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
  }
}
```

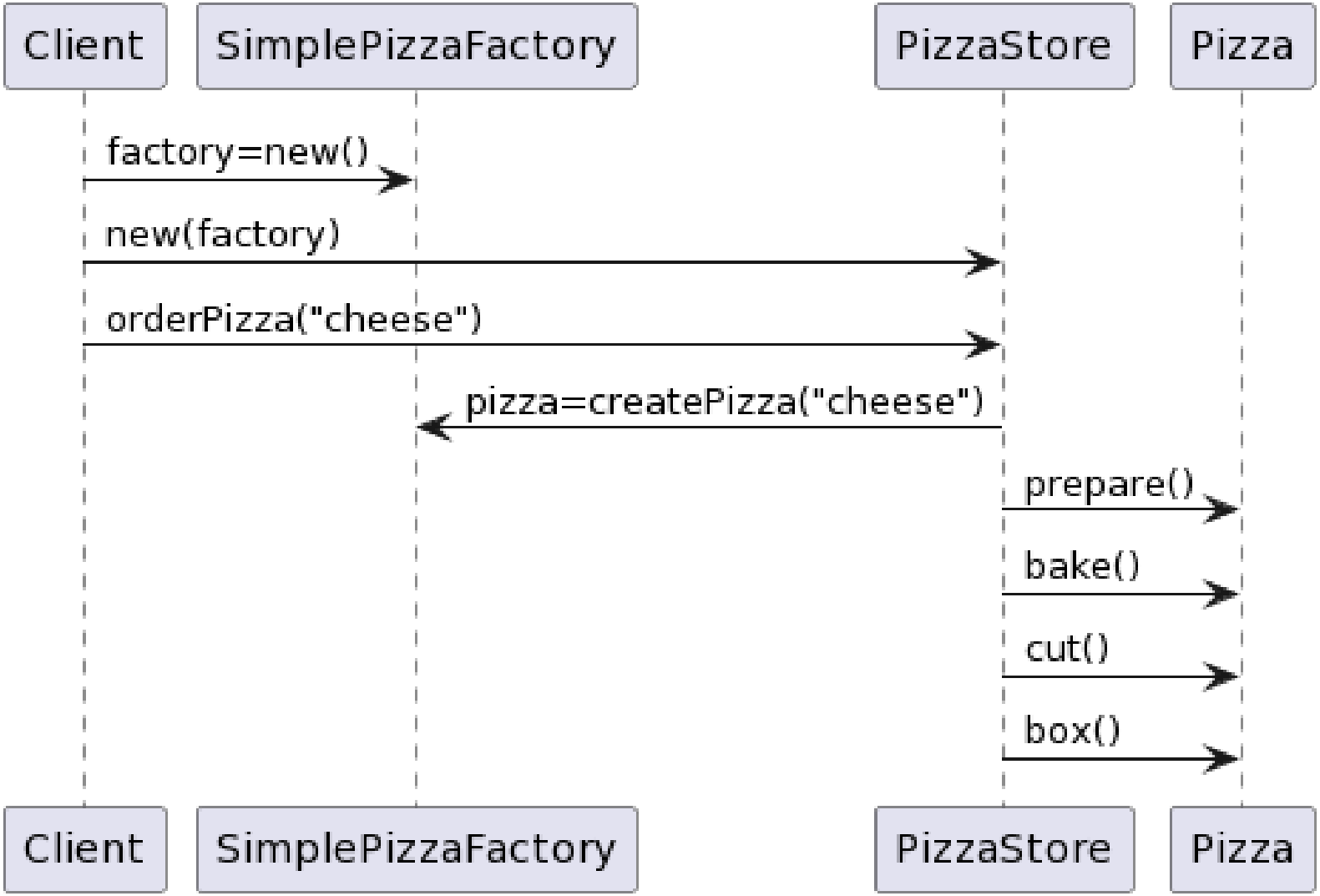
Pizza Store (HFDP Ch. 4)

- PizzaStore Class Diagram



Each Pizza class implements Pizza.

Pizza Store (HFDP Ch. 4)

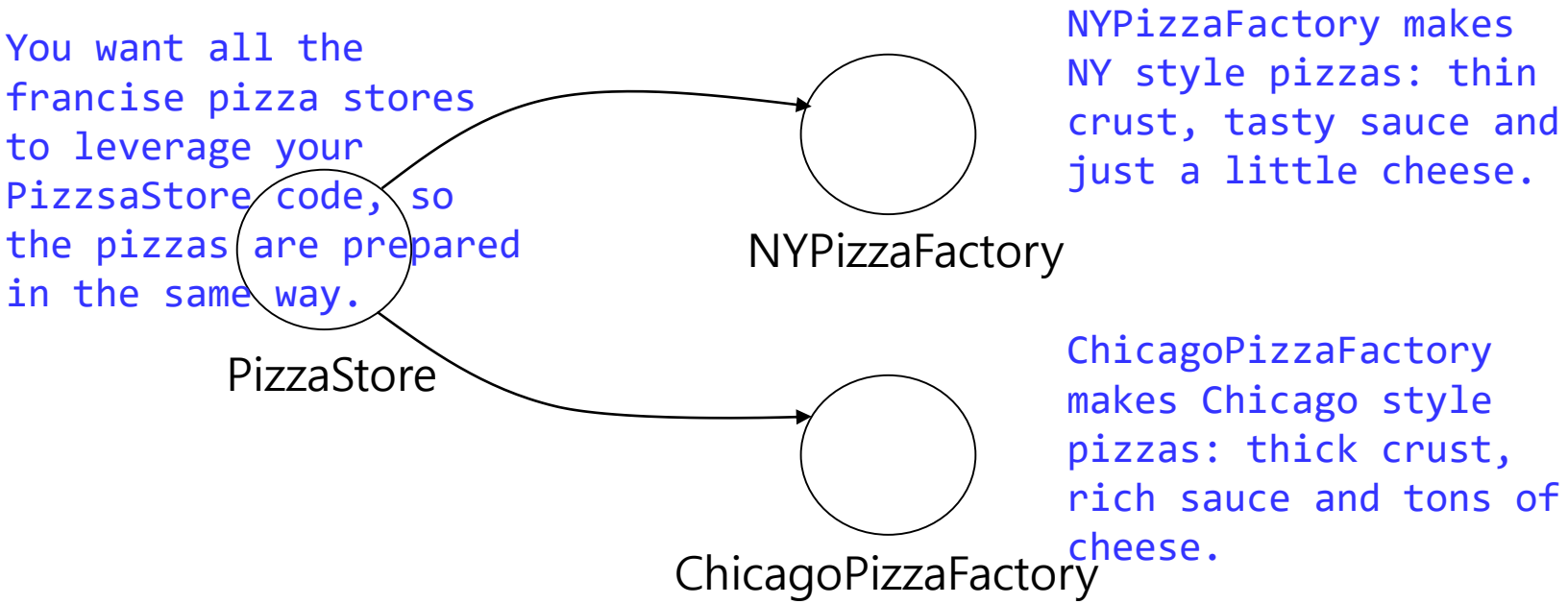


Simple Factory

- 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)

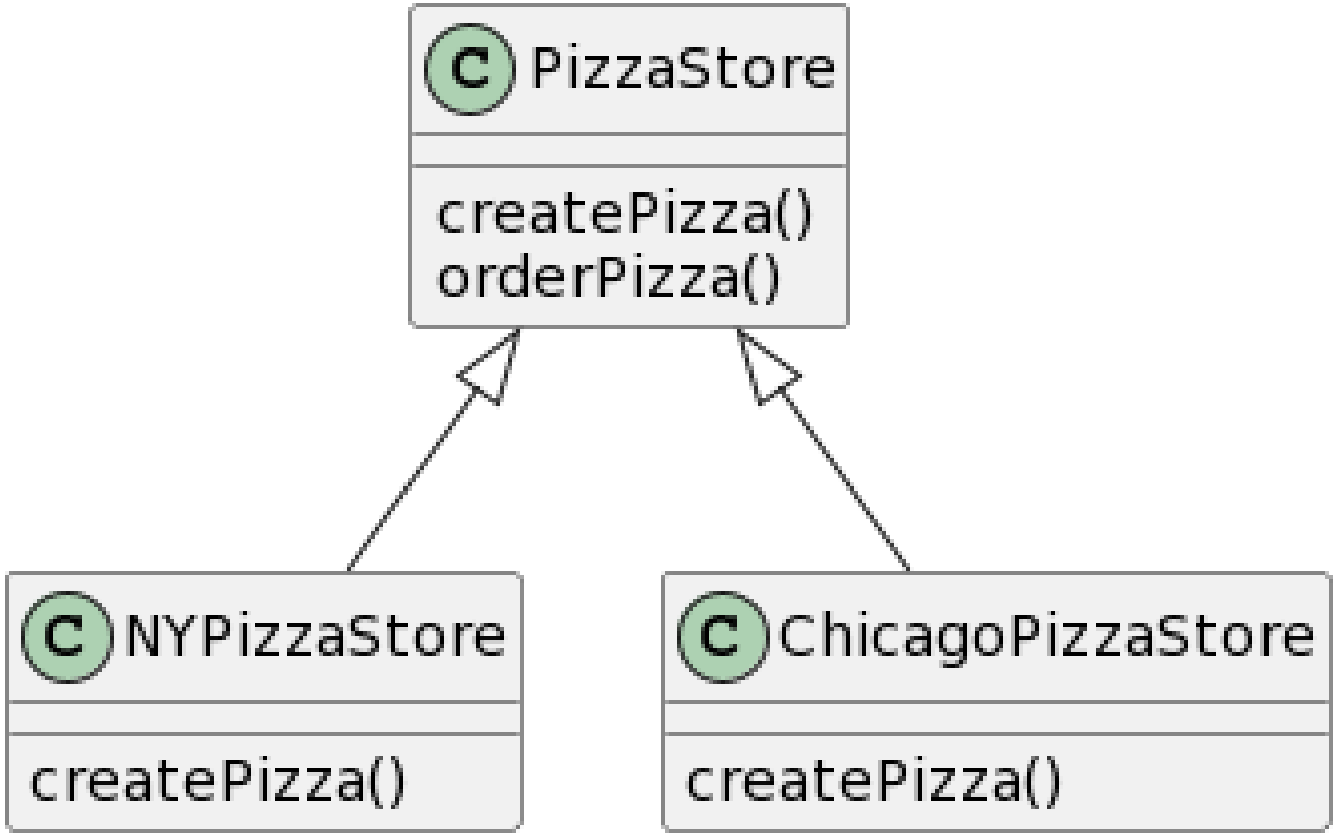
```
public abstract class PizzaStore {
  void prepareToBoxing(Pizza pizza) {
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
  }

  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;
  }
}
```

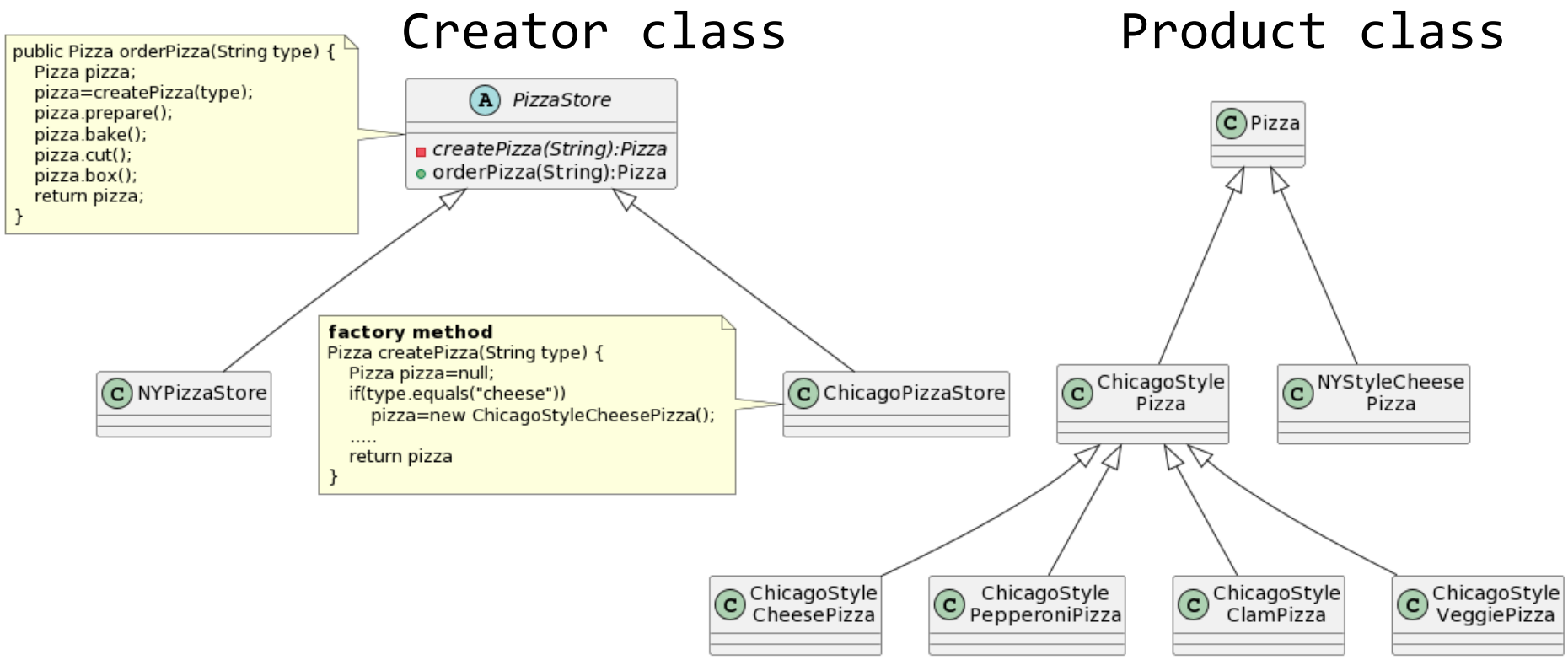
```
public class NYStyleCheesePizza extends Pizza {
  public NYStyleCheesePizza() {
    name = "NY Style Sauce and Cheese Pizza";
    dough = "Thin Crust Dough";
    sauce = "Marinara Sauce";
    toppings.add("Grated Reggiano Cheese");
  }
}

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");
  }
}
```

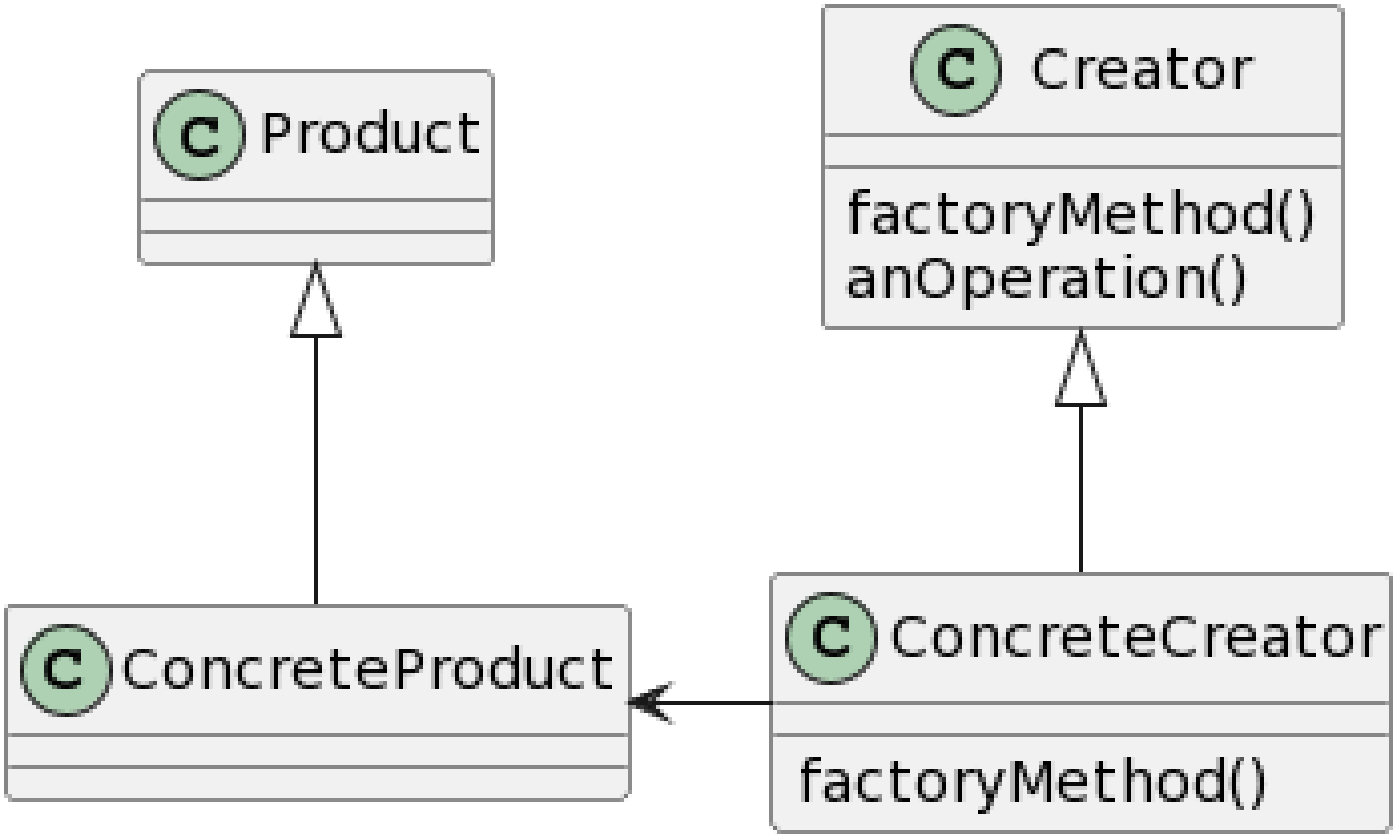
main method

```
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

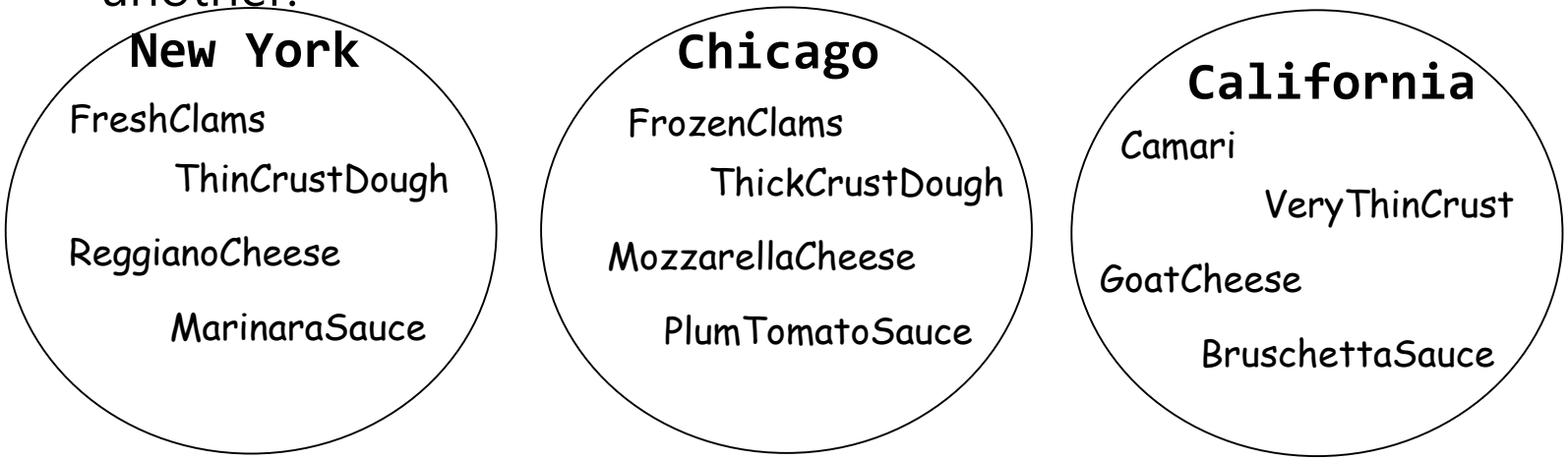
- Creator
  - Defines a method that needs to create an object whose actual type is unknown. Does so using abstract method call.
- ConcreteCreator
  - Subclass that overrides the abstract object-instantiation method to create the Concrete Product.
- Product
  - Interface implemented by the created product. 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();
  }
}
```

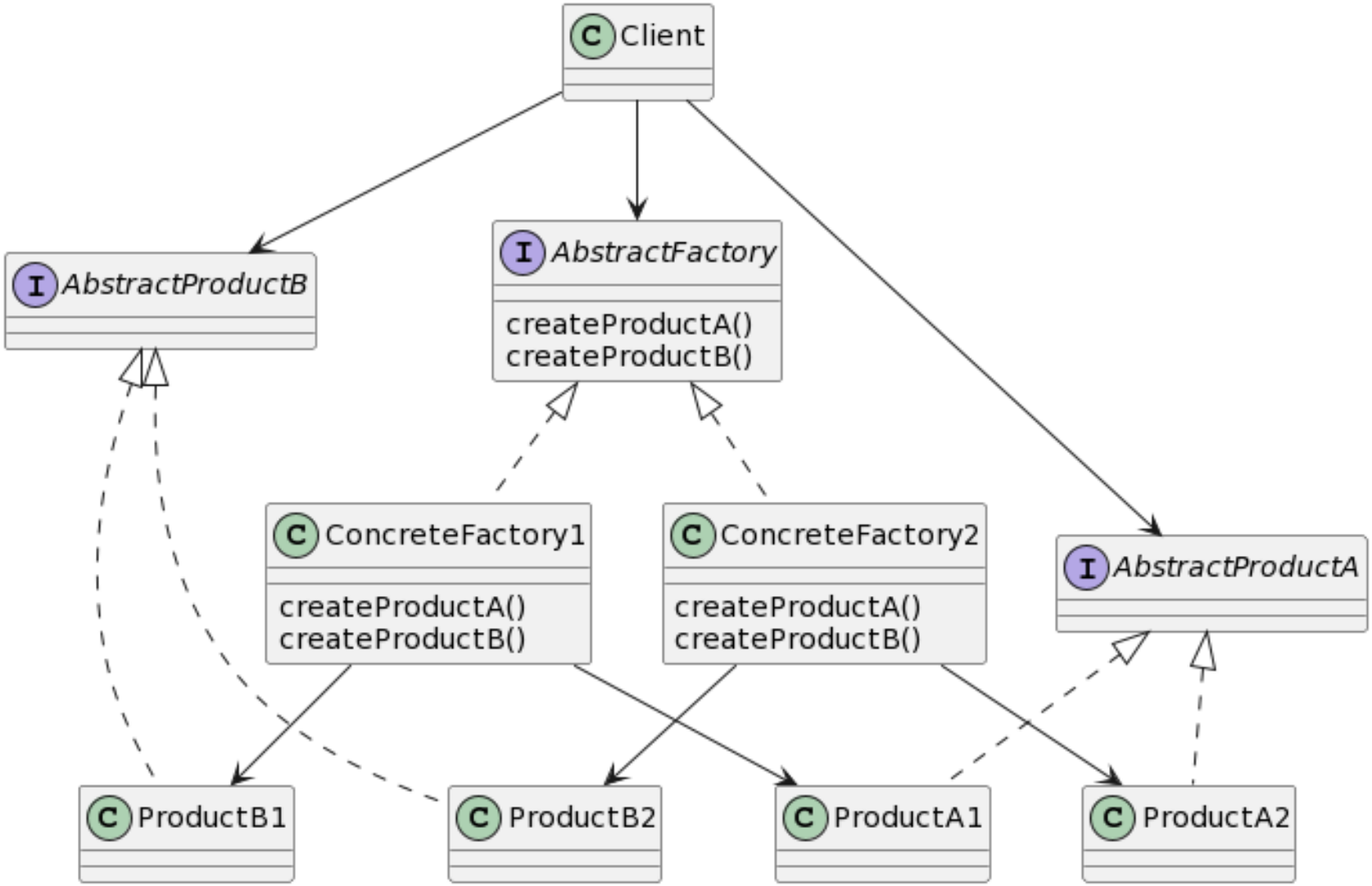
Families of Pizza Ingredients

```
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 specifies 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

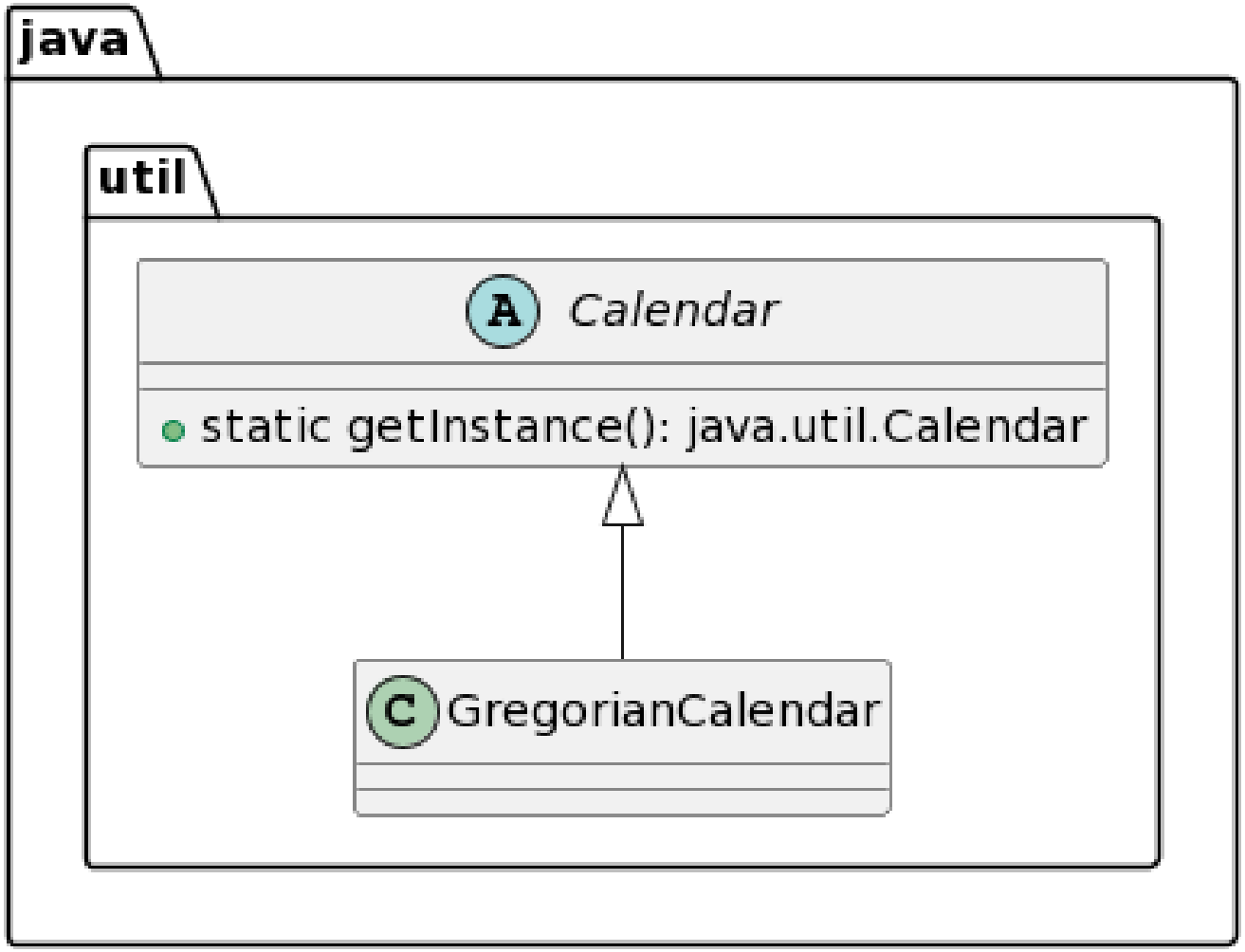
Abstract Factory Pattern



Abstract Factory Pattern

- AbstractFactory
  - Defines the interface that all concrete factories must implement, which consists of a set of methods for creating products.
- ConcreteFactory1, ConreteFactory2
  - Each concrete factory can product 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



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.

Builder Pattern

- Aims to "Separate the construction of a complex object from its representation so that the same construction process can create different representations".
- It is used to construct a complex object step by step and the final step will return the object.
- The builder pattern should be used when we want to build different ***immutable objects*** using the *same object building process*.
- The only big **difference between the builder pattern and the abstract factory pattern** is that builder provides us more control over the object creation process, and that's it.

Builder Pattern

- java.util.Appendable
- java.lang.StringBuilder#append() [unsynchronized class]
- java.lang.StringBuffer#append() [synchronized class]
- java.nio.ByteBuffer#put() (also on CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer and DoubleBuffer)
- javax.swing.GroupLayout.Group@addComponent()
- Lombok's @Builder annotation is a useful technique to implement the builder pattern.

Problem

- Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects.
- Such initialization code is usually buried inside a monstrous constructor with lots of parameters.
- What if only *bun* and *patty* are *mandatory*, and the *rest* are *optional*. We need more constructors. This problem is called the **telescoping constructor problem.**
  - public Burger(**int bun, int patty**, boolean cheese, boolean lettuce, boolean tomato, boolean bacon) { … }
  - public Burger(**int bun, int patty**, boolean cheese, boolean lettuce, boolean tomato) { … }
  - public Burger(**int bun, int patty**, boolean cheese, boolean lettuce { … }
  - public Burger(**int bun, int patty**, boolean cheese) { … }
  - public Burger(**int bun, int patty**) { … }

Problem

- Problem with telescoping constructor
  - Making the constructor calls pretty ugly.

```
  // all ingredient
  Burger burger1 = new Burger(2, 1, true, true,
true, true);
  // bun, patty2, cheese
  Burger burger2 = new Burger(2, 2, true);
  // bun, patty, bacon
  Burger burger3 = new Burger(2, 1, false, false,
false, true);
```

- Now let's add more field in the Burger class.
  - Problem! One way is to create more constructors, and another is to lose the immutability and introduce setter methods. You choose any of both options, and you lose something.
- The Builder pattern help you to consume additional fields while retaining the immutability of the class.

Builder Pattern

	Description
Pattern	Builder
Problem	Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects. Such initialization code is usually buried inside a monstrous constructor with lots of parameters. *You might make the program too complex by creating a subclass for every possible configuration of an object. Or, The constructor with lots of parameters has its downside: not all the parameters are needed at all times.*
Solution	The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built.
Result	OCP, SRP

Burger

```
public class Burger {
  private int bun; // required
  private int patty; // required
  private boolean cheese; // optional
  private boolean lettuce; // optional
  private boolean tomato; // optional
  private boolean bacon; // optional

  public Burger(int bun, int patty, boolean
cheese, boolean lettuce, boolean tomato,
boolean bacon) { … }
  public Burger(int bun, int patty, boolean
cheese, boolean lettuce, boolean tomato) { … }
  public Burger(int bun, int patty, boolean
cheese, boolean lettuce) { … }
  …
}
```

Telescoping constructors problem

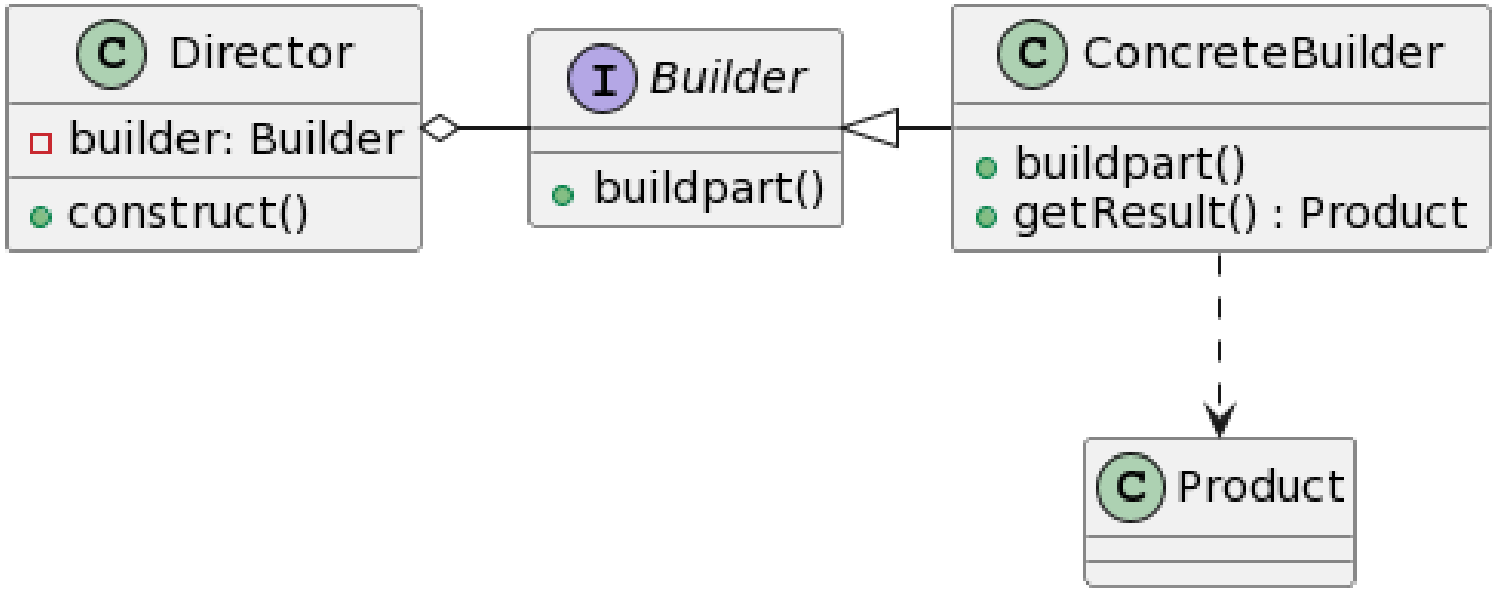
Burger

```
public class Burger {
  private int bun; // required
  private int patty; // required
  private boolean cheese; // optional
  private boolean lettuce; // optional
  private boolean tomato; // optional
  private boolean onion; // optional
  private boolean bacon; // optional

  public Burger(int bun, int patty, boolean
cheese, boolean lettuce, boolean tomato,
boolean onion, boolean bacon) { … }
  public Burger(int bun, int patty, boolean
cheese, boolean lettuce, boolean tomato,
boolean onion) { … }
…
}
```

Telescoping constructors problem

Builder Pattern



Define Builder Pattern

- Builder
  - declares product construction steps that are common to all types of builders.
- ConcreteBuilder
  - provides different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface.
- Product
  - is an resulting object. Products constructed by different builders don't have to belong to the same class hierarchy or interface.
- Director
  - defines the order in which to call construction steps, so you can create and reuse specific configurations of products.

Burger

- **BurgerBuilder** help us in building desired instance with all required fields and a combination of optional fields.

```
public class Burger {
  private final int bun; // required
  private final int patty; // required
  private final boolean cheese; // optional
  private final boolean lettuce; // optional
  private final boolean tomato; // optional
  private final boolean bacon; // optional
  private Burger(BurgerBuilder builder) {
    this.bun = builder.bun;
    this.patty = builder.patty;
    this.cheese = builder.cheese;
    this.lettuce = builder.lettuce;
    this.tomato = builder.tomato;
    this.bacon = builder.bacon;
  }
```

Burger

```
// all getter, and no setter to provide immutability
public int getBun() {
  return bun;
}
public int getPatty() {
  return patty;
}
public boolean getCheese() {
  return cheese;
}
public boolean getLettuce() {
  return lettuce;
}
… // getTomato(), getBacon() 중간 생략
@Override
public String toString() {
  …
}
```

Burger

```
  // BurgerBuilder
  public static class BurgerBuilder {
    private final int bun; // required
    private final int patty; // required
    private boolean cheese; // optional
    private boolean lettuce; // optional
    private boolean tomato; // optional
    private boolean bacon; // optional

    public BurgerBuilder(int bun, int patty) {
      this.bun = bun;
      this.patty = patty;
    }
    public BurgerBuilder cheese(boolean cheese) {
      this.cheese = cheese;
      return this;
    }
… // lettuce, tomato 중간 생략
```

Burger

```
    // BurgerBuilder
    public boolean bacon(boolean bacon) {
      this.bacon = bacon;
      return this;
    }
    public Burger build() {
      return new Burger(this);
    }
  } // end of BurgerBuilder class
} // end of Burger class
```

Burger

```
public static void main(String[] args) {
  Burger burger1 = new Burger.BurgerBuilder(2,1)
      .cheese(true)
      .lettuce(true)
      .tomato(true)
      .bacon(true)
      .build();
  System.out.println(burger1);
  // bun, patty2, cheese
  Burger burger2 = new Burger.BurgerBuilder(2,2)
      .cheese(true)
      .build(); // no lettuce, tomato, bacon
  System.out.println(burger2);
  // bun, patty, bacon
  Burger burger3 = new Burger.BurgerBuilder(2,2)
      .bacon(true)
      .build(); // no cheese, lettuce, tomato
  System.out.println(burger3);
```