Factories

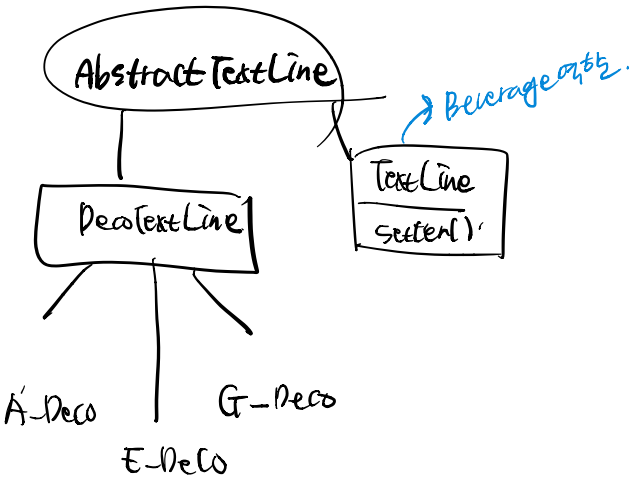
Simple Factory

Factory Method

Abstract Factory

Young → * → *young*

= → =young=



new를 개선하고자 Creational → factory

What's wrong with "new"?

객체에 대한 변화..

바뀔수가 없다.

- When you see "new", think "concrete".

```
Duck duck = new MallardDuck();
```

We want to use interfaces to keep code flexible.

But we have to create an instance of a concrete class!

```
Duck duck;

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

We have a bunch of different duck classes, and we don't know until runtime which one we need to instantiate.

- when you have code that makes use of lots of concrete classes, you're looking for trouble because that code may have to be changed as new concrete classes are added.

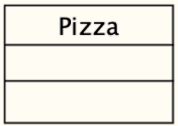
high maintenance cost !

- Violating OCP

생성자를 없애고 싶은 이유

간략히 다시 언급해 보면, 상속. 특히 자바에서는 인터페이스를 이용하여 객체의 변화를 반영하는 좋은 방법이 있었지만, 이 것 역시 생성자의 한계를 벗어날 수 없어 다른 코드의 변화로부터 독립적인 코드의 작성에 생성자가 걸림돌이 되었다. 생성자는 각각 확정된 객체 (Concrete object)가 있어야 호출될 수 있으며, 확정된 객체에 변화가 생기면 잠재적으로 수정될 가능성이 생긴다. 물론 여기에 있는 예제에서처럼 생성자가 쉽게 찾아 볼 수 있다면 편하겠지만, 수만, 수천 줄의 코드에 생성자 호출이 여기저기에 펼쳐져 있으면 그 수만, 수천 줄의 코드를 모두 살펴야 한다. 따라서 생성자를 따로 관리 할 수 있다면 좀 더 변화에 대응하기가 쉽다.

Pizza shop example

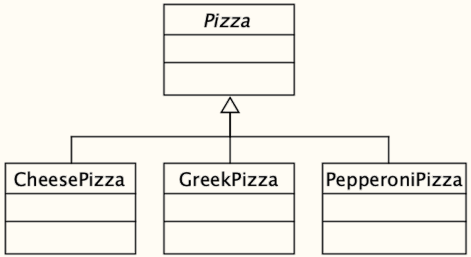



```
Pizza orderPizza() {
        Pizza pizza = new Pizza();

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
}
```

For flexibility, we really want this to be an abstract class or interface, but we can't directly instantiate either of those.

- If you need more than one type of pizza...

Creating mass of “new” for flexibility




```
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();
        }

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
}
```

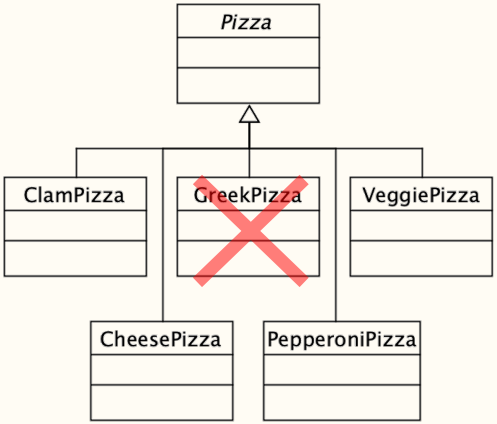
We're now passing in the type of pizza to orderPizza.

Based on the type of pizza, we instantiate the correct concrete class and assign it to the pizza instance variable. Note that each pizza here has to implement the Pizza interface.

Once we have a Pizza, we prepare it (you know, roll the dough, put on the sauce and add the toppings & cheese), then we bake it, cut it and box it!

Each Pizza subtype (CheesePizza, VeggiePizza, etc.) knows how to prepare itself.

When changes are made…



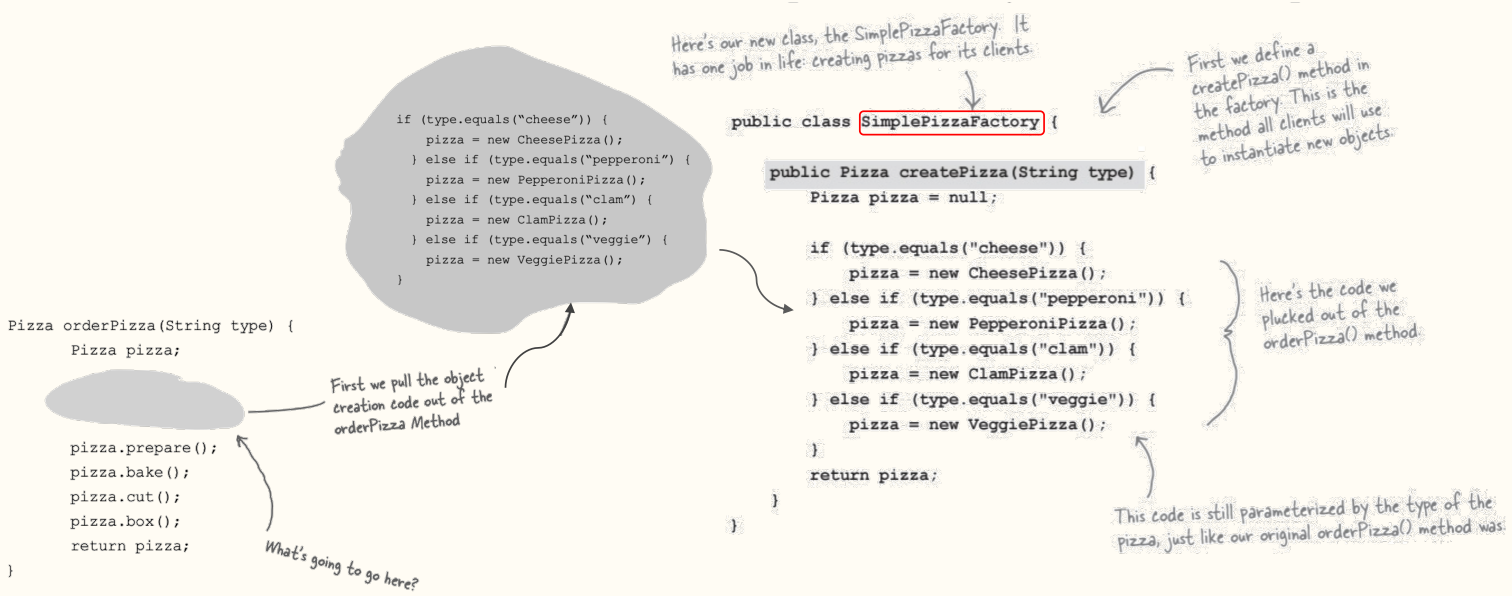
This code is NOT closed for modification. If the Pizza Shop changes its pizza offerings, we have to get into this code and modify it.

```
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();
    }

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

This is what varies. As the pizza selection changes over time, you'll have to modify this code over and over.

This is what we expect to stay the same. For the most part, preparing, cooking, and packaging a pizza has remained the same for years and years. So, we don't expect this code to change, just the pizzas it operates on.

Extracting varies (Encapsulating object instantiation)



Put them together !

```
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;
    }
}
```

This code

```
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    // other methods here
}
```

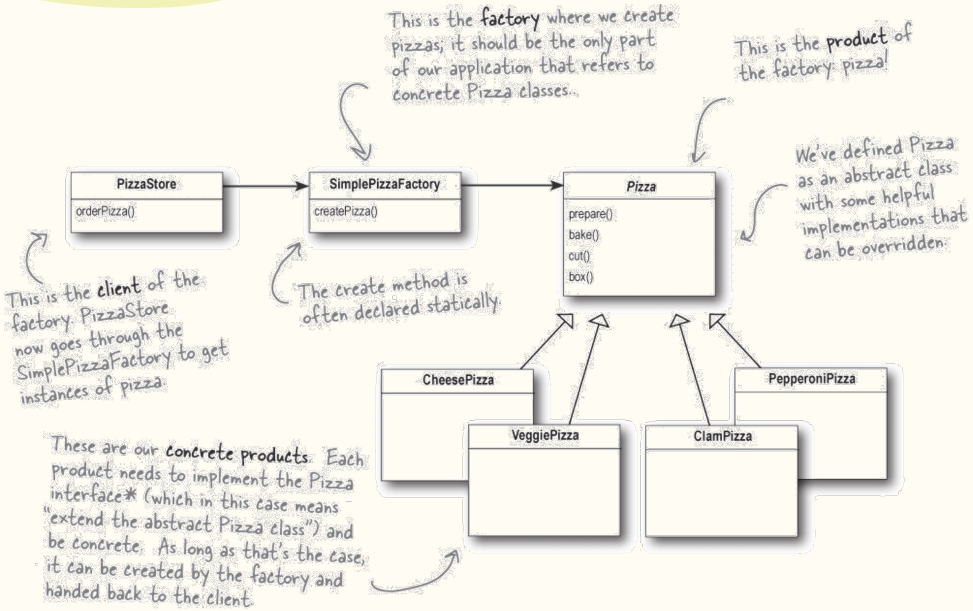
Now we give PizzaStore a reference to a SimplePizzaFactory.

PizzaStore gets the factory passed to it in the constructor.

And the orderPizza() method uses the factory to create its pizzas by simply passing on the type of the order.

Notice that we've replaced the **new operator** with a create **method** on the factory object. No more concrete instantiations here!

Simple Factory ~~Pattern~~ Idiom?!



Advantage of simple factory

- By encapsulating the pizza creating in one class, we now have only one place to make modifications when the implementation changes.

- Introducing **Static Factory**
  - Just make SimpleFactory static (class)
  - Pros:
    - No need to instantiate an object to make use of the create method
  - Cons:
    - Can’t make a subclass -> no extension
    - Can’t change the behavior of the create method

Factory Method

Creational Pattern

Dealing variation while keeping the process

process는 동일하지만 결과는 다를때 ☆

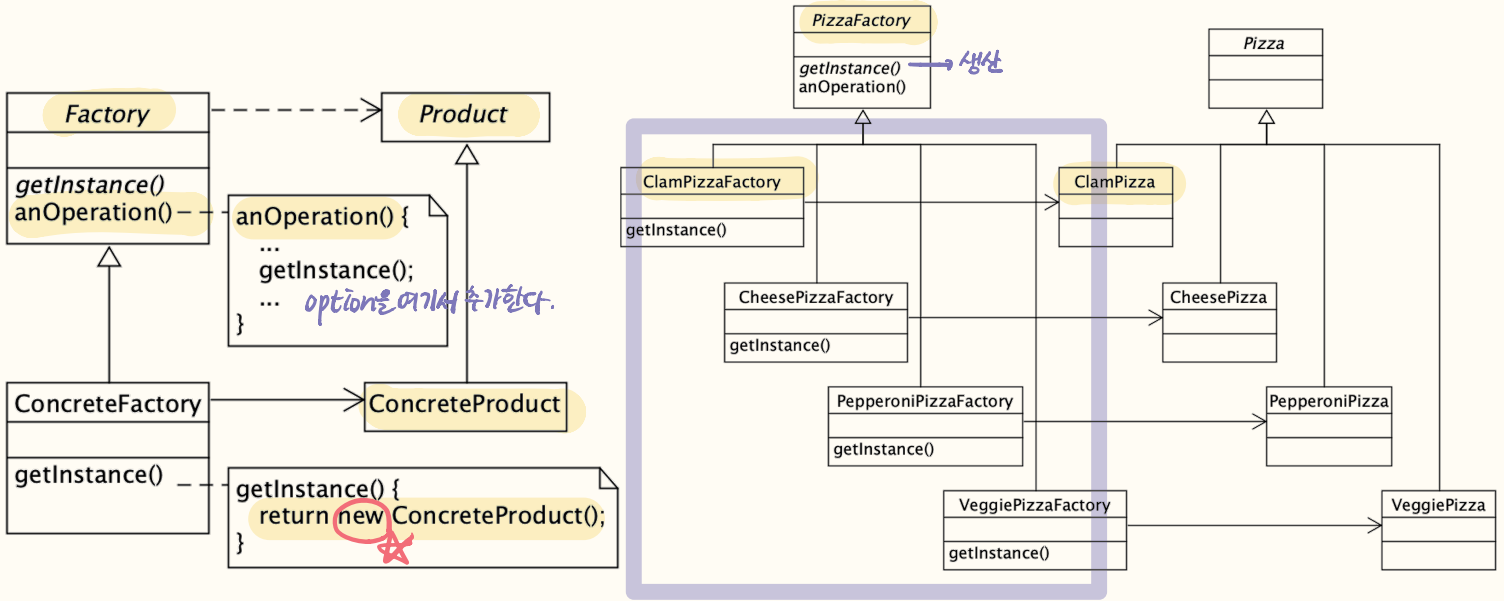
- Since createPizza() method has a process of preparing pizza and an instantiation of pizza, varying part can be separated from fixed part.
- One part (varying part) is an instantiation of pizza, and the other part () is preparing pizza.
- Instancing which pizza should be determined in subclass (for the choice variety)
- Let's name each part;
  - varying part = getInstance()
  - fixed part = preparePizza()

← createPizza

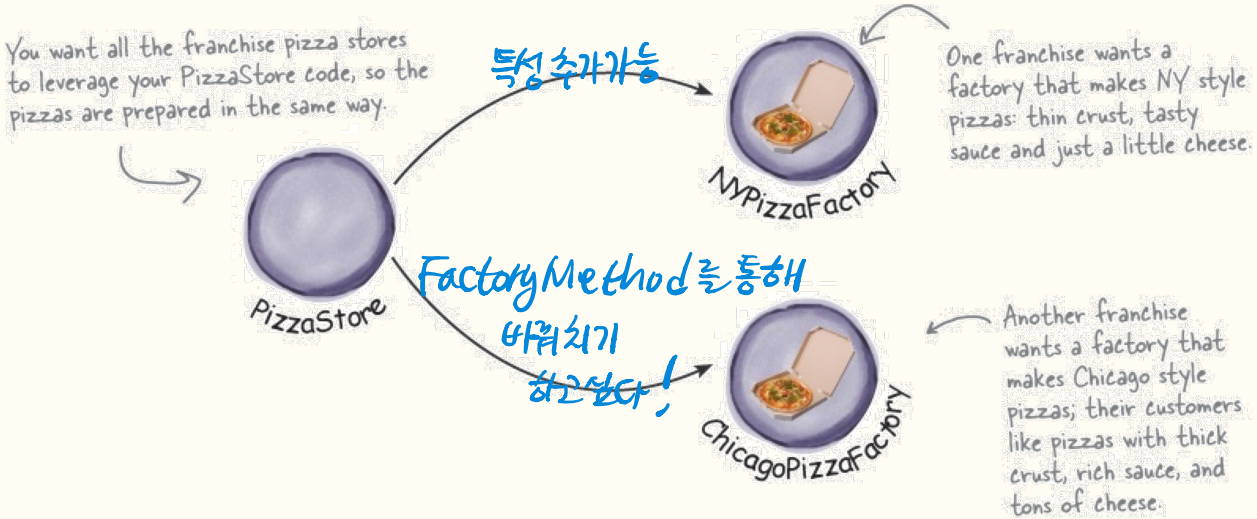
Factory Method

- **Intention**: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **Motivation**: How can an object be created so that subclasses can redefine which class to instantiate? How can a class defer instantiation to subclasses? (delaying choice of the class)
- **Solution**: Define a separate abstract operation (factory method) for creating an object. Create an object by calling a factory method (Since factory method is abstract, creating instance is deferred to subclass).

Structure of factory method



Preparing Franchise



Apply factory method (Franchise as a factory)

Simple Factory 로 구현하다.

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

Method 생성 과정을 거쳐

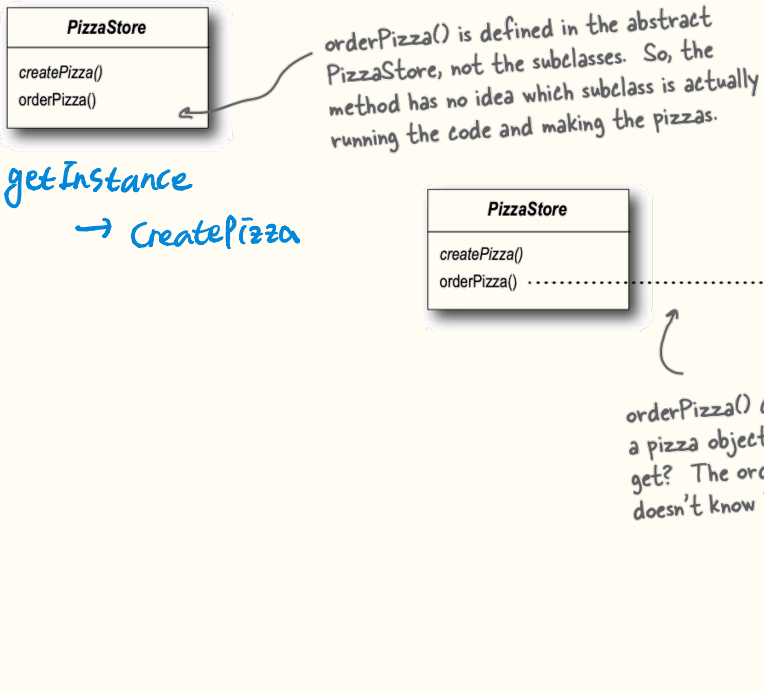
Here we create a factory for making NY style pizzas.

Then we create a PizzaStore and pass it a reference to the NY factory.

...and when we make pizzas, we get NY style pizzas.

```
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.orderPizza("Veggie");
```

Likewise for the Chicago pizza stores: we create a factory for Chicago pizzas and create a store that is composed with a Chicago factory. When we make pizzas, we get the Chicago style ones.


PizzaStore
createPizza()
orderPizza()
orderPizza() is defined in the abstract PizzaStore, not the subclasses. So, the method has no idea which subclass is actually running the code and making the pizzas.
getInstance
→ CreatePizza
PizzaStore
createPizza()
orderPizza()
pizza = createPizza();
pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
fixed part
orderPizza() calls createPizza() to actually get a pizza object. But which kind of pizza will it get? The orderPizza() method can't decide; it doesn't know how. So who does decide?
varying part
NYStylePizzaStore
createPizza()
ChicagoStylePizzaStore
createPizza()

PizzaStore is now abstract (see why below).

```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    abstract Pizza createPizza(String type);
}
```

Our "factory method" is now abstract in PizzaStore.

Each subclass provides an implementation of the createPizza() method, overriding the abstract createPizza() method in PizzaStore, while all subclasses make use of the orderPizza() method defined in PizzaStore. We could make the orderPizza() method final if we really wanted to enforce this.

If a franchise wants NY style pizzas for its customers, it uses the NY subclass, which has its own createPizza() method, creating NY style pizzas.

Similarly, by using the Chicago subclass, we get an implementation of createPizza() with Chicago ingredients.

생성이 Pizza생성

Remember: createPizza() is abstract in PizzaStore, so all pizza store subtypes MUST implement the method.

```
public Pizza createPizza(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();
    }
}
```

Simple Factory Code 고려중

```
public Pizza createPizza(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();
    }
}
```

1 **Let's follow Ethan's order: first we need a NY PizzaStore:**

```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

Creates a instance of NYPizzaStore

2 **Now that we have a store, we can take an order:**

```
nyPizzaStore.orderPizza("cheese");
```

The orderPizza() method is called on the nyPizzaStore instance (the method defined inside PizzaStore runs)

createPizza("cheese")

3 **The orderPizza() method then calls the createPizza() method:**

```
Pizza pizza = createPizza("cheese");
```

Remember, createPizza(), the factory method, is implemented in the subclass. In this case it returns a NY Cheese Pizza

4 **Finally, we have the unprepared pizza in hand and the orderPizza() method finishes preparing it:**

```
pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
```

All of these methods are defined in the specific pizza returned from the factory method createPizza(), defined in the NYPizzaStore

The orderPizza() method gets back a Pizza, without knowing exactly what concrete class it is

Adding detail

We'll start with an abstract Pizza class and all the concrete pizzas will derive from this

Each Pizza has a name, a type of dough, a type of sauce, and a set of toppings

The abstract class provides some basic defaults for baking, cutting and boxing

Preparation follows a number of steps in a particular sequence

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

    void prepare() {
        System.out.println("Preparing " + name);
        System.out.println("Tossing dough...");
        System.out.println("Adding sauce...");
        System.out.println("Adding toppings: ");
        for (String topping : toppings) {
            System.out.println("   " + topping);
        }
    }

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

    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");
    }
}
```

The NY Pizza has its own marinara style sauce and thin crust.

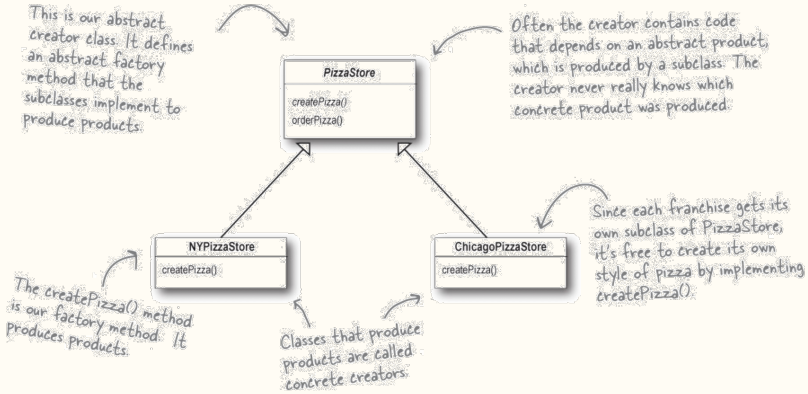
And one topping, reggiano cheese!

The Chicago Pizza uses plum tomatoes as a sauce along with extra-thick crust.

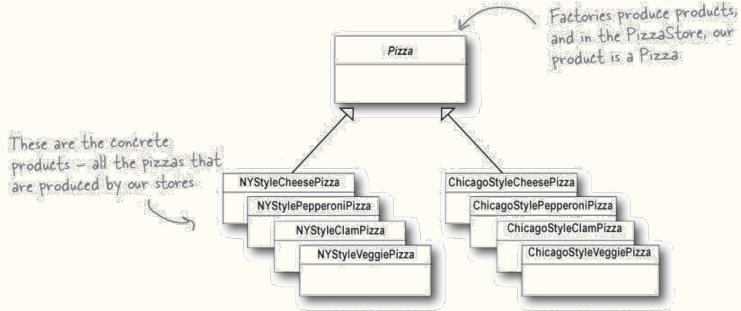
The Chicago style deep dish pizza has lots of mozzarella cheese!

The Chicago style pizza also overrides the cut() method so that the pieces are cut into squares.

The Creator classes



The Product classes

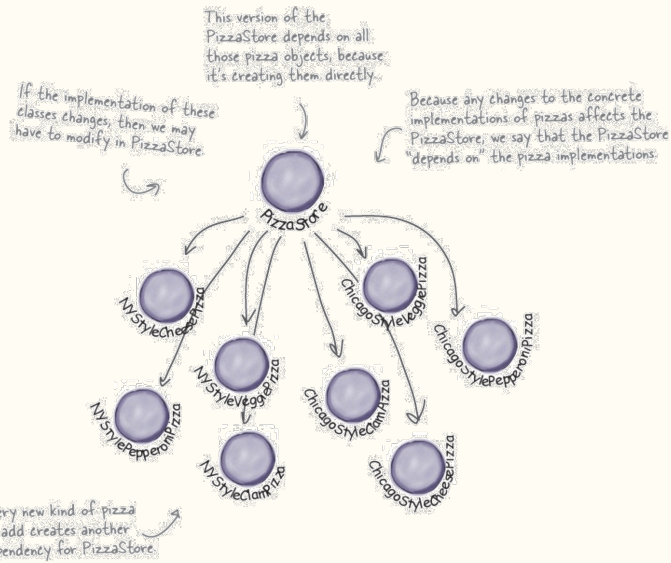


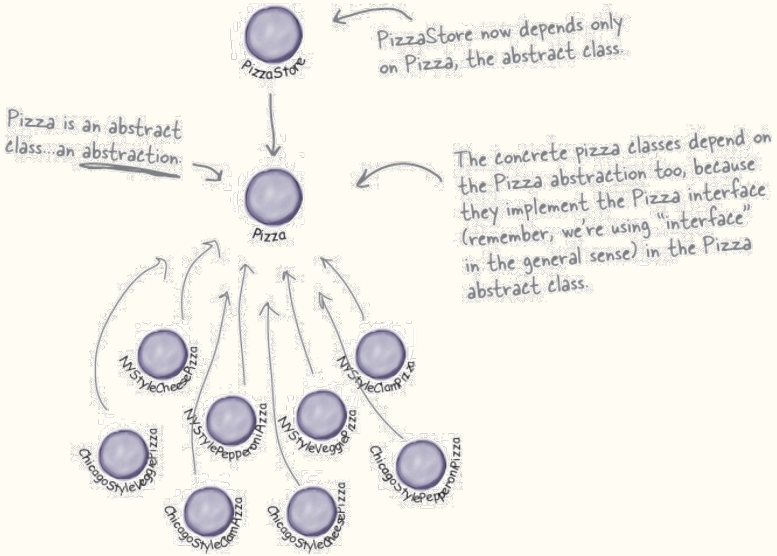
Abstract Factory

Creational Pattern

Dependency Inversion Principal

High level components should not depend on our low-level components; rather they should both depend on abstractions.



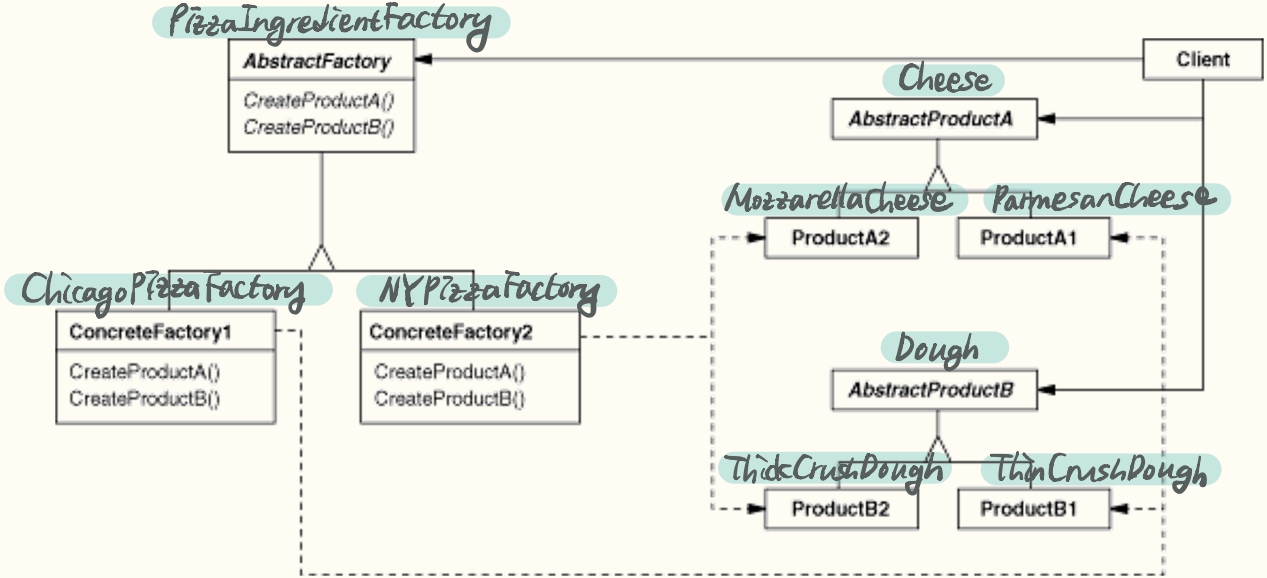


Ingredients Problem

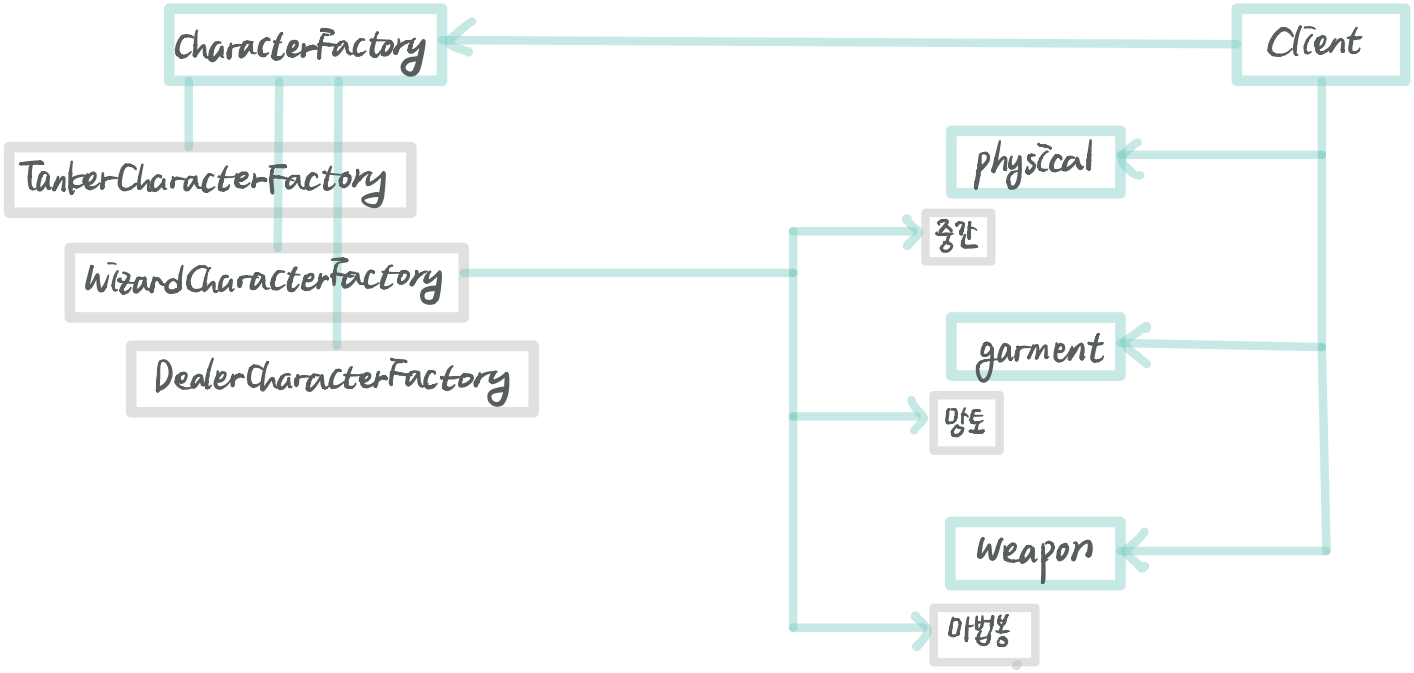
Abstract Factory Pattern

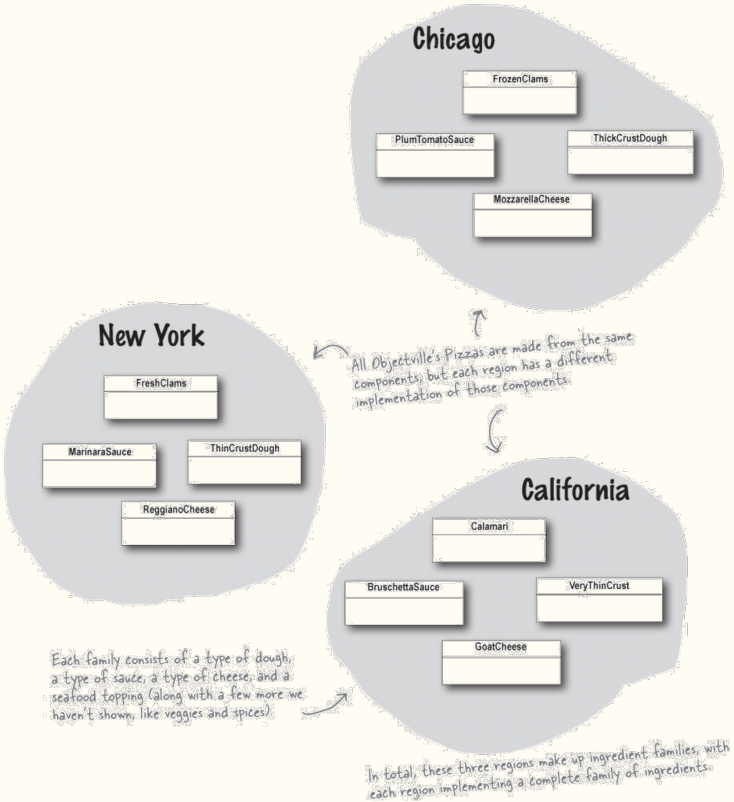
- **Intention**: provide an interface for creating families of related or dependent objects without specifying their concrete classes
- **Motivation** : Families of related objects need to be instantiated
- **Solution**: Coordinates the creation of families of objects. Gives a way to take the rules of how to perform the instantiation out of the client
- **Consequence**:
  - Requires extending the Factory interface to extend an object family.
  - Introduces an additional level of indirection.

Structure


PizzaIngredientFactory
AbstractFactory
CreateProductA()
CreateProductB()
Client
Cheese
AbstractProductA
MozzarellaCheese
ParmesanCheese
ProductA2
ProductA1
ChicagoPizzaFactory
NYPizzaFactory
ConcreteFactory1
CreateProductA()
CreateProductB()
ConcreteFactory2
CreateProductA()
CreateProductB()
Dough
AbstractProductB
ThickCrustDough
ThinCrustDough
ProductB2
ProductB1

Character 객체 생성하는
Abstract Character Factory





Apply abstract factory

```
public interface PizzaIngredientFactory {

    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();

}
```

Lots of new classes here, one per ingredient.

```
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;
    }

    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }

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

For each ingredient in the ingredient family, we create the New York version.

For veggies, we return an array of Veggies. Here we've hardcoded the veggies. We could make this more sophisticated, but that doesn't really add anything to learning the factory pattern, so we'll keep it simple.

The best sliced pepperoni. This is shared between New York and Chicago. Make sure you use it on the next page when you get to implement the Chicago factory yourself

New York is on the coast; it gets fresh clams. Chicago has to settle for frozen.

```
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");
    }

    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() {
        // code to print pizza here
    }
}
```

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

We've now made the prepare method abstract. This is where we are going to collect the ingredients needed for the pizza, which of course will come from the ingredient factory.

Our other methods remain the same, with the exception of the prepare method.

```
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();
    }
}
```

To make a pizza now, we need a factory to provide the ingredients. So each Pizza class gets a factory passed into its constructor, and it's stored in an instance variable.

Here's where the magic happens!

The prepare() method steps through creating a cheese pizza, and each time it needs an ingredient, it asks the factory to produce it.

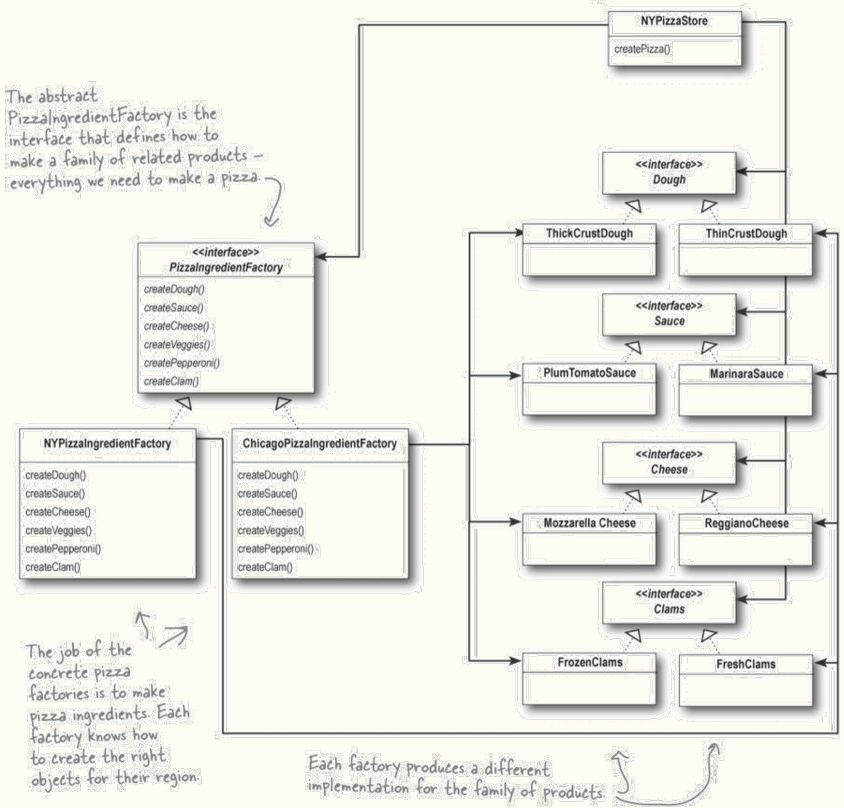
```
public class NYPizzaStore extends PizzaStore {

    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")) {

            pizza = new ClamPizza(ingredientFactory);
            pizza.setName("New York Style Clam Pizza");

        } else if (item.equals("pepperoni")) {

            pizza = new PepperoniPizza(ingredientFactory);
            pizza.setName("New York Style Pepperoni Pizza");

        }
        return pizza;
    }
}
```

The NY Store is composed with a NY pizza ingredient factory. This will be used to produce the ingredients for all NY style pizzas.

We now pass each pizza the factory that should be used to produce its ingredients.

Look back one page and make sure you understand how the pizza and the factory work together!

For each type of Pizza, we instantiate a new Pizza and give it the factory it needs to get its ingredients.


NYPizzaStore
createPizza()
The abstract PizzaIngredientFactory is the interface that defines how to make a family of related products – everything we need to make a pizza
<<interface>>
PizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
<<interface>>
Dough
ThickCrustDough
ThinCrustDough
<<interface>>
Sauce
PlumTomatoSauce
MarinaraSauce
<<interface>>
Cheese
Mozzarella Cheese
ReggianoCheese
<<interface>>
Clams
FrozenClams
FreshClams
NYPizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
ChicagoPizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
The job of the concrete pizza factories is to make pizza ingredients. Each factory knows how to create the right objects for their region.
Each factory produces a different implementation for the family of products.

Abstract Factory

abstract class Pizza

- String name;
- Dough dough;
- Veggies veggies[];
- Cheese cheese;
- Pepperoni pepperoni;
- Clams clam;

abstract void prepare();
void bake();
void cut();
void box();
void setName(); → this.name = name;
getName();

public String toString();

public abstract class PizzaStore

abstract Pizza createPizza(String item);
public Pizza orderPizza(String type);
↳ Pizza pizza = createPizza(type);
pizza.prepare();
bake();
cut();
box();
return pizza;

factory import
Pizza
PizzaStore
Main

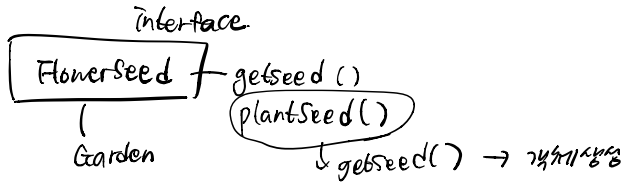
Main

PizzaStore store = new NYPizzaStore();

Pizza pizza = store.orderPizza("cheese");

pizza = new CheesePizza(ingredientFactory);
pizza.setName("~");

PizzaIngreFac ingredientFactory;
public CheesePizza( )
void prepare(){
dough = ingredientFactory.createDough();
Sauce = ingredientFactory.createSauce();
cheese = ingredientFactory.createCheese();



<Talent> → Fighter

<RoyalFighter> → Fighter

Sward / Armor 장착