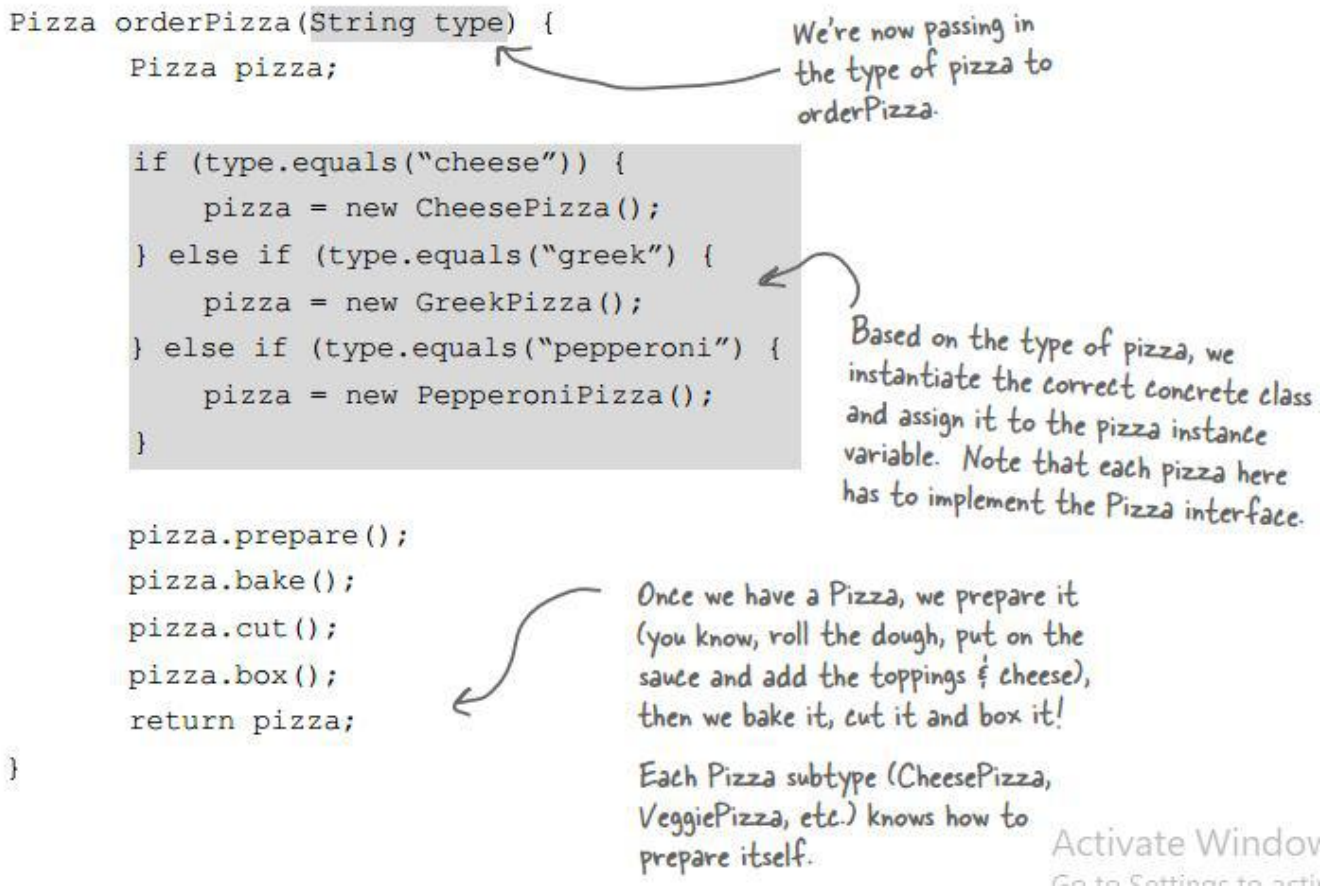
Factory Pattern

Mainly there are two main type of factory pattern:

1)Simple factory pattern
2)Factory Method
3)Abstract Factory Pattern

***Problem:**

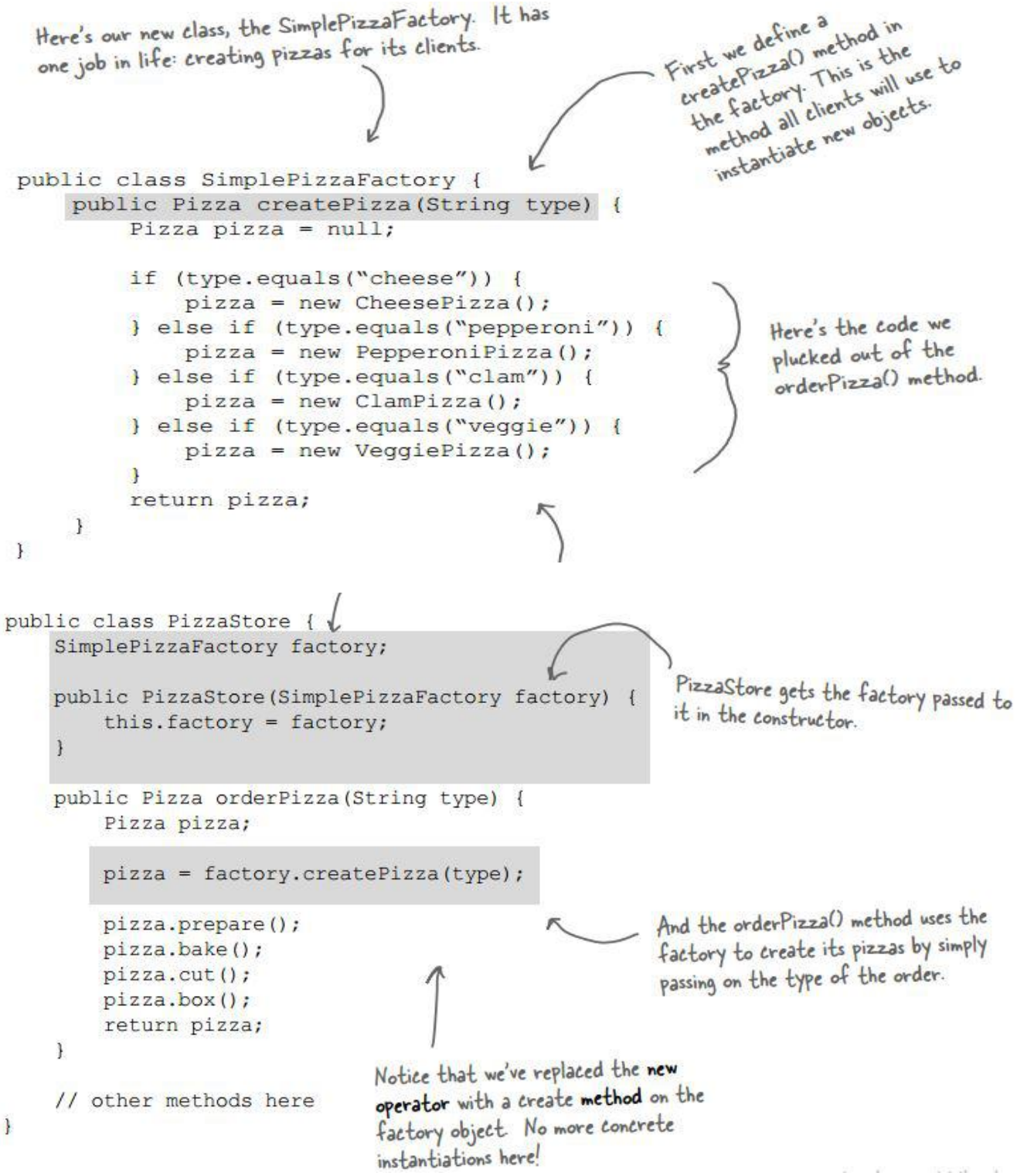


1)The Simple Factory:

Clearly, dealing with which concrete class is instantiated is really messing up our orderPizza() method and preventing it from being closed for modification. But now that we know what is varying and what isn't, it's probably time to encapsulate it.

So now we know we'd be better off moving the object creation out of the orderPizza() method. But how? Well, what we're going to do is take the creation code and move it out into another object that is only going to be concerned with creating pizzas.We've got a name for this new object: we call it a **Factory**.Factories handle the details of object creation.

Factory Pattern




```
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;
    }
}
```

```
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
}
```

Factory Pattern

2)Factory Method Pattern:

The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

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

        return pizza;
    }

    protected abstract Pizza createPizza(String type);

    // other methods here
}
```

The subclasses of PizzaStore handle object instantiation for us in the createPizza() method.

All the responsibility for instantiating Pizzas has been moved into a **method** that acts as a **factory**.

Code Up Close

A factory method handles object creation and encapsulates it in a subclass. This decouples the client code in the superclass from the object creation code in the subclass.

A factory method may be parameterized (or not) to select among several variations of a product.

```
abstract Product factoryMethod(String type)
```

A factory method is abstract so the subclasses are counted on to handle object creation.

A factory method returns a Product that is typically used within methods defined in the superclass.

A factory method isolates the client (the code in the superclass, like orderPizza()) from knowing what kind of concrete Product is actually created.

All factory patterns encapsulate object creation. The Factory Method Pattern encapsulates object creation by letting subclasses decide what objects to create.

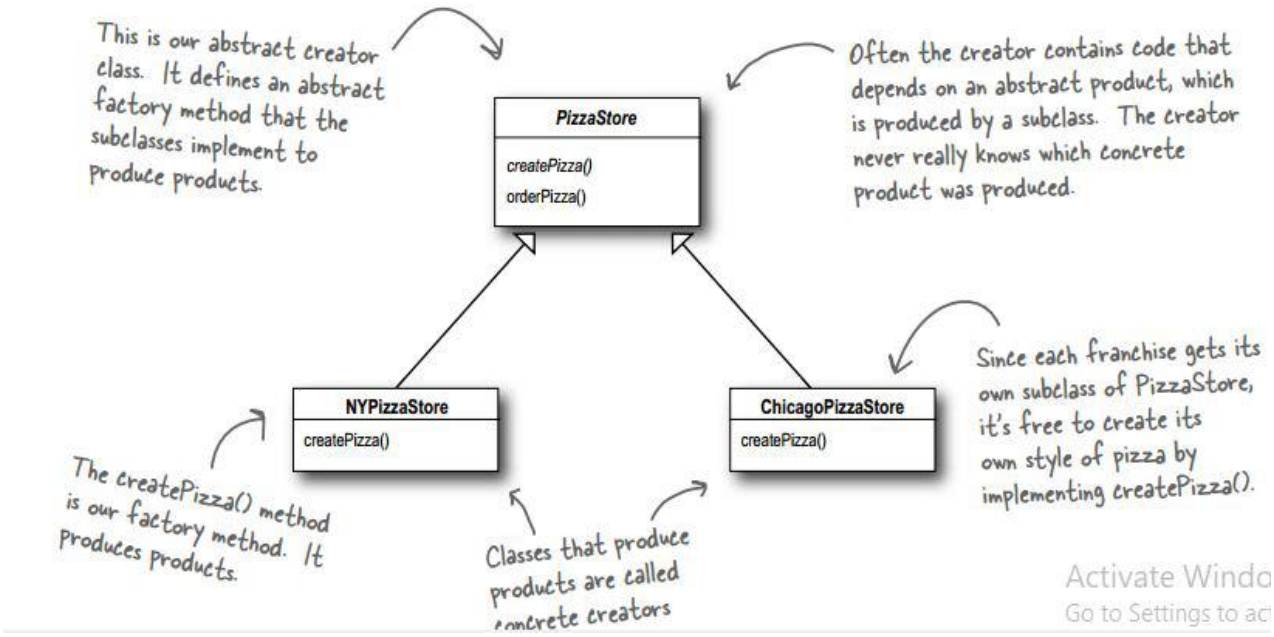
Factory Pattern

```
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");
    }
}
```

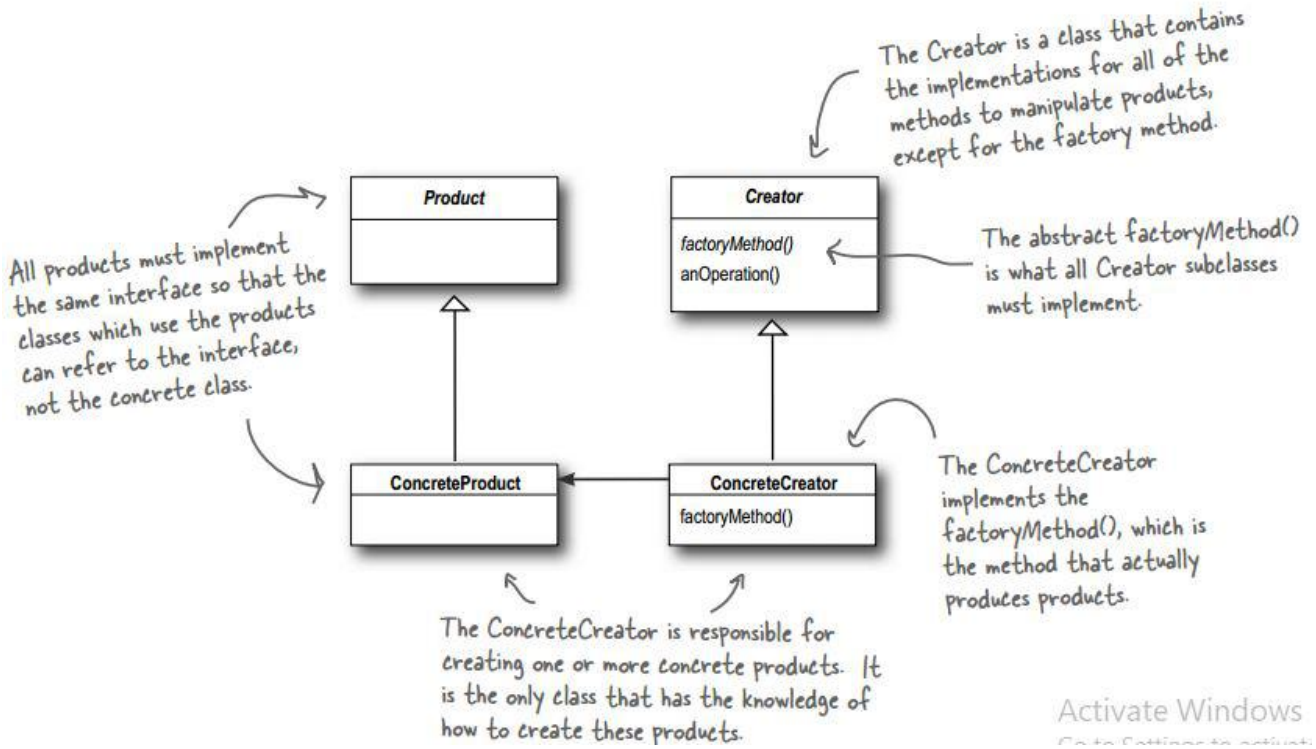
First we create two different stores.

Then use one one store to make Ethan's order.

The Creator classes



Factory Pattern



By placing all my creation code in one object or method, I avoid duplication in my code and provide one place to perform maintenance. That also means clients depend only upon interfaces rather than the concrete classes required to instantiate objects. As I have learned in my studies, this allows me to program to an interface, not an implementation, and that makes my code more flexible and extensible in the future.

Factory Pattern

The Dependency Inversion Principle:

When you directly instantiate an object, you are depending on its concrete class.

Design Principle
Depend upon abstractions. Do not depend upon concrete classes.

It suggests that our high-level components should not depend on our low-level components; rather, they should both depend on abstractions.

```
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 (type.equals("clam")) {
                pizza = new NYStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new NYStylePepperoniPizza();
            }
        } else if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
                pizza = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new ChicagoStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new ChicagoStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new ChicagoStylePepperoniPizza();
            }
        } else {
            System.out.println("Error: invalid type of pizza");
            return null;
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
```

Handles all the NY style pizzas

Handles all the Chicago style pizzas

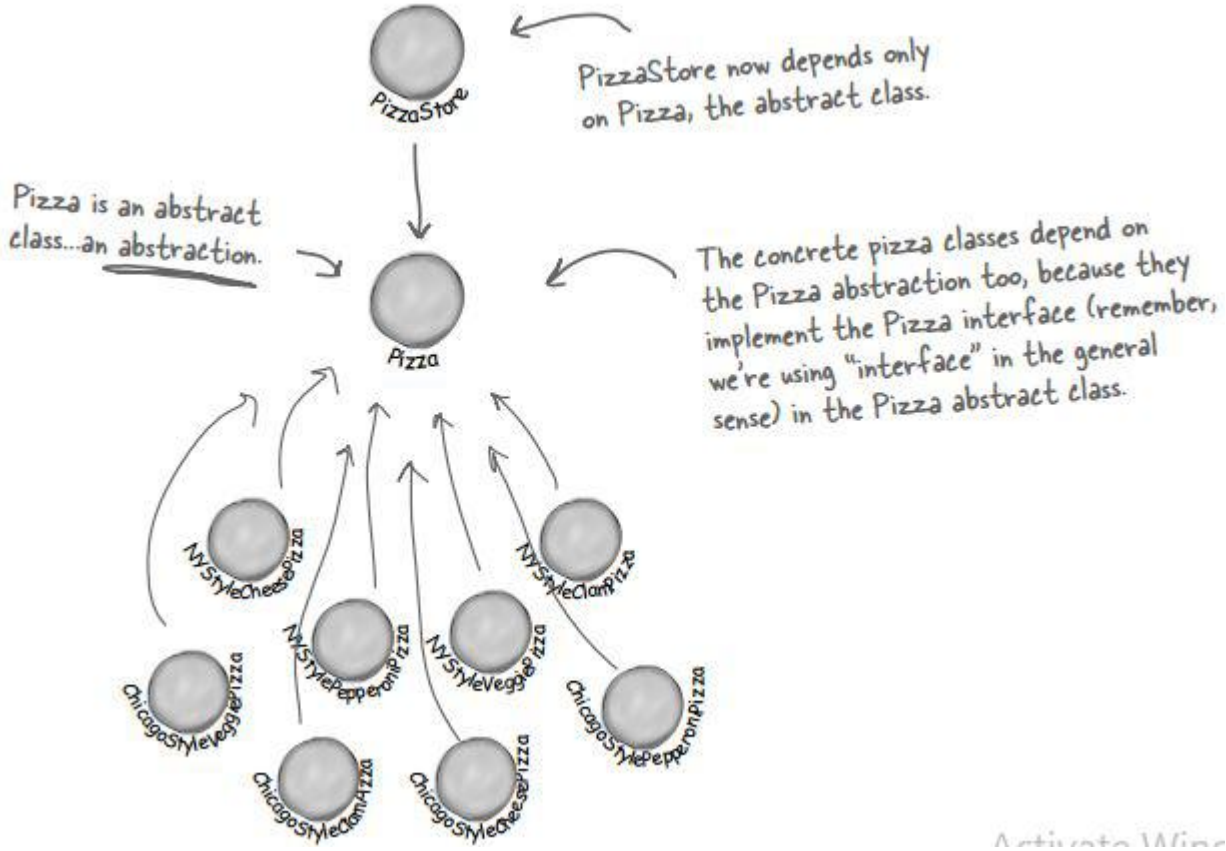
A "high-level" component is a class with behavior defined in terms of other, "low level" components. For example, PizzaStore is a high-level component because its behavior is defined in terms of pizzas - it creates all the different pizza objects, prepares, bakes, cuts, and boxes them, while the pizzas it uses are low-level components

Factory Pattern

The "inversion" in the name Dependency Inversion Principle is there because it inverts the way you typically might think about your OO design.

The following guidelines can help you avoid OO designs that violate the Dependency Inversion Principle:

- No variable should hold a reference to a concrete class.
- No class should derive from a concrete class.
- No method should override an implemented method of any of its base classes.



This is a guideline you should strive for, rather than a rule you should follow all the time.

After applying the Factory Method, you'll notice that our high-level component, the PizzaStore, and our low-level components, the pizzas, both depend on Pizza, the abstraction. Factory Method is not the only technique for adhering to the Dependency Inversion Principle, but it is one of the more powerful ones.

Factory Pattern

3)Abstract Factory Pattern:

For Pizza example:
Now we're going to build a factory to create our ingredients; the factory will be responsible for creating each ingredient in the ingredient family.

Here's what we're going to do:

- Build a factory for each region. To do this, you'll create a subclass of PizzaIngredientFactory that implements each create method.
- Implement a set of ingredient classes to be used with the factory, like ReggianoCheese, RedPeppers, and ThickCrustDough. These classes can be shared among regions where appropriate.

```
public interface PizzaIngredientFactory {

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

}
```

For each ingredient we define a create method in our interface.

Lots of new classes here, one per ingredient.

If we'd had some common "machinery" to implement in each instance of factory, we could have made this an abstract class instead...

- Then we still need to hook all this up by working our new ingredient factories into our old PizzaStore code.

Factory Pattern

```
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.

```
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");
    }
```

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.

Factory Pattern

```
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.

Code Up Close

The Pizza code uses the factory it has been composed with to produce the ingredients used in the pizza. The ingredients produced depend on which factory we're using. The Pizza class doesn't care; it knows how to make pizzas. Now, it's decoupled from the differences in regional ingredients and can be easily reused when there are factories for the Rockies, the Pacific Northwest, and beyond.

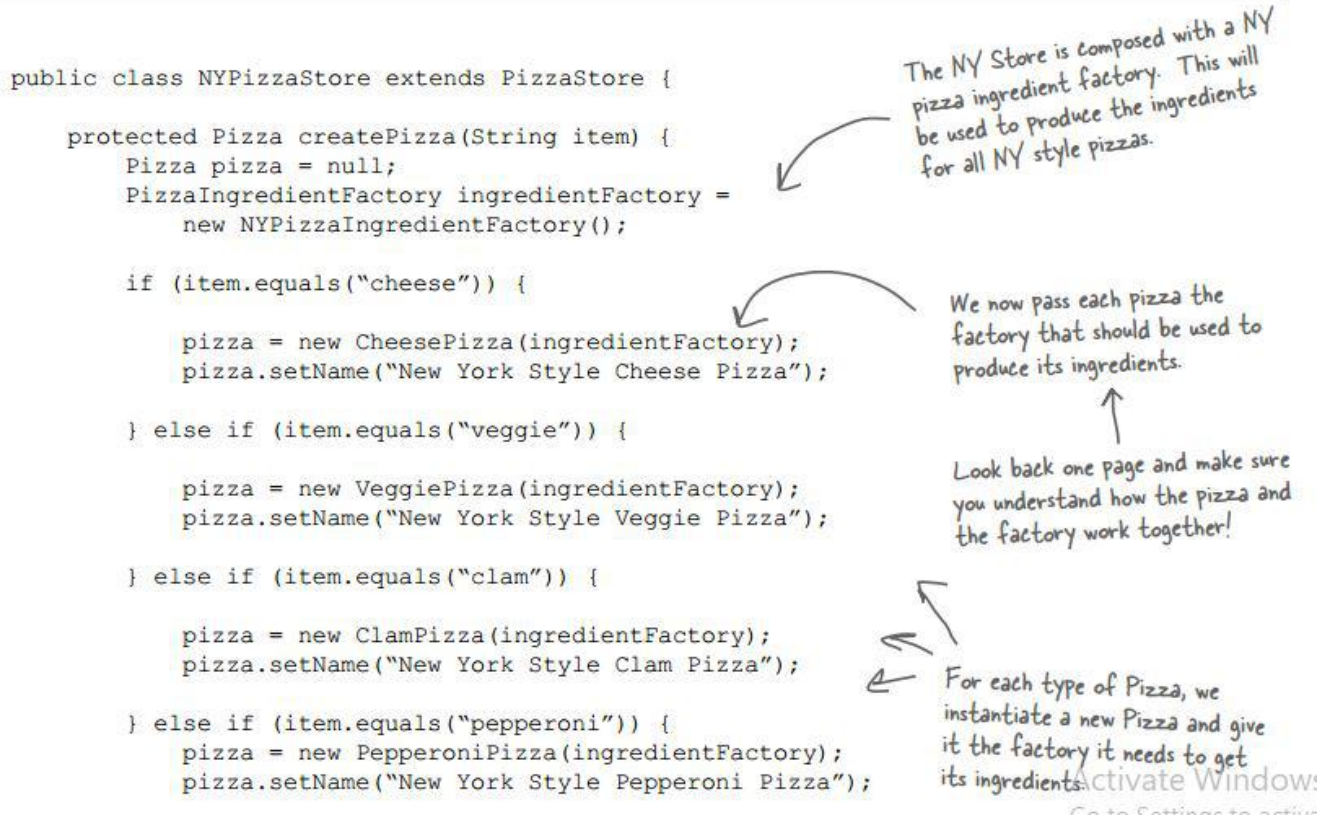
```
sauce = ingredientFactory.createSauce();
```

We're setting the Pizza instance variable to refer to the specific sauce used in this pizza.

This is our ingredient factory. The Pizza doesn't care which factory is used, as long as it is an ingredient factory.

The createSauce() method returns the sauce that is used in its region. If this is a NY ingredient factory, then we get marinara sauce.

Factory Pattern



An Abstract Factory gives us an interface for creating a family of products. By writing code that uses this interface, we decouple our code from the actual factory that creates the products. That allows us to implement a variety of factories that produce products meant for different contexts – such as different regions, different operating systems, or different look and feels. Because our code is decoupled from the actual products, we can substitute different factories to get different behaviors (like getting marinara instead of plum tomatoes).

The Abstract Factory Pattern :

The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

We've certainly seen that Abstract Factory allows a client to use an abstract interface to create a set of related products without knowing (or caring) about the concrete products that are actually produced. In this way, the client is decoupled from any of the specifics of the concrete products.The job of an Abstract Factory is to define an interface for creating a set of products. Each method in that interface is responsible for creating a concrete product, and we implement a subclass of the Abstract Factory to supply those implementations. So, factory methods are a natural way to implement your product methods in your abstract factories.

Factory Pattern

Factory Method	Abstract Factory
I use classes to create.	I use objects.
I do it through inheritance.	I do it through object composition.
use me to decouple your client code from the concrete classes you need to instantiate, or if you don't know ahead of time all the concrete classes you are going to need. To use me, just subclass me and implement my factory method!	use me whenever you have families of products you need to create and you want to make sure your clients create products that belong together.
	I provide an abstract type for creating a family of products. Subclasses of this type define how those products are produced. To use the factory, you instantiate one and pass it into some code that is written against the abstract type.
	I need a big interface because I am used to create entire families of products. You're only creating one product, so you don't really need a big interface, you just need one method.

---

1)Simple Factory pattern:
    Create class and put object creation login in it.
2)Factory Method:
    Create abstract class and abstract method in it to create objects.
3)Abstract Factory pattern: