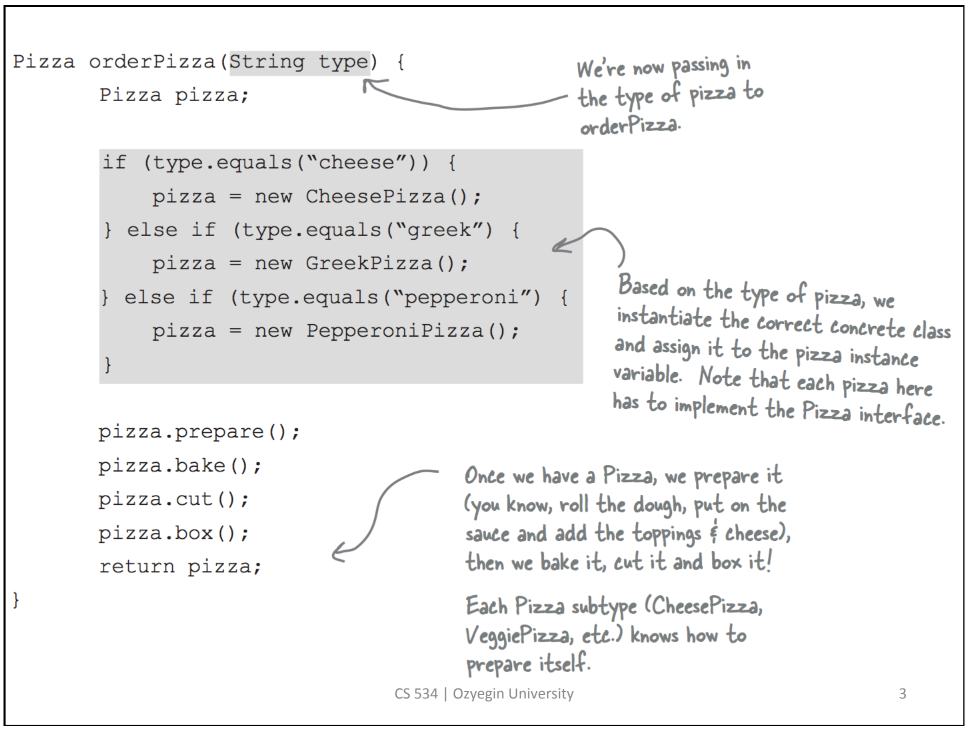
Abstract Factory

Creational Patterns

- Hide how instances of classes are created and put together
- Flexibility in what gets created, who creates it, how it gets created, and when.



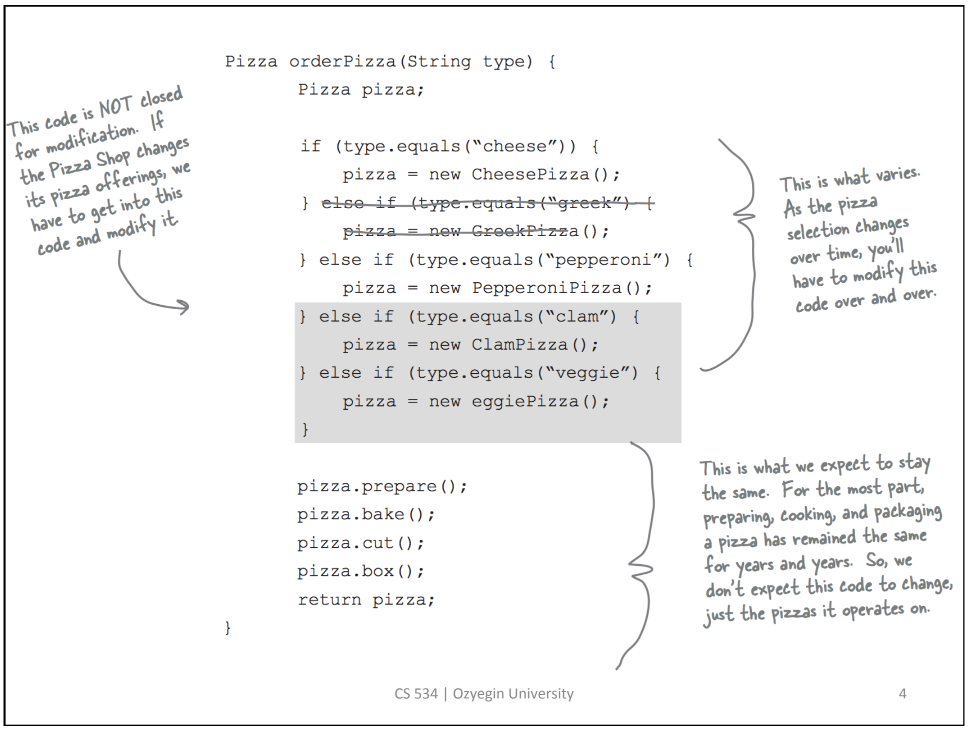
```
Pizza orderPizza(String type) {
        Pizza pizza;

        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.



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 eggiePizza();
        }

        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.

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



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

First we pull the object creation code out of the orderPizza Method

Then we place that code in an object that is only going to worry about how to create pizzas. If any other object needs a pizza created, this is the object to come to.

```
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;
    }
}
```

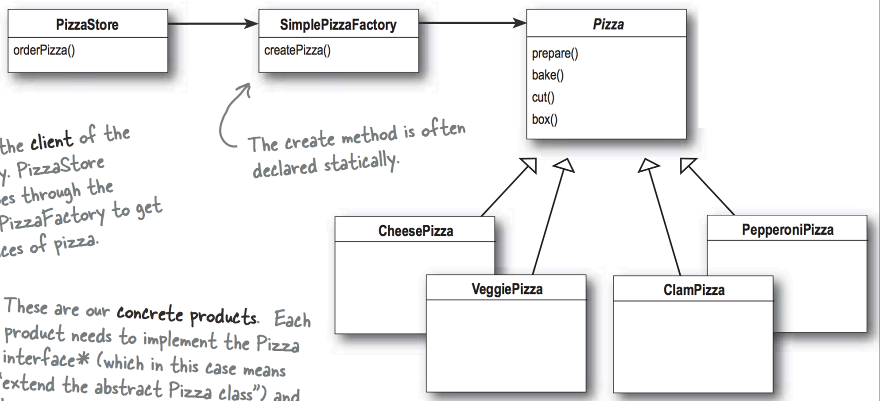
Now we give PizzaStore a reference to a SimplePizzaFactory.

```
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
}
```

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!



This is the **client** of the factory. PizzaStore now goes through the SimplePizzaFactory to get instances of pizza.

The create method is often declared statically.

These are our **concrete products**. Each product needs to implement the Pizza interface* (which in this case means "extend the abstract Pizza class") and be concrete. As long as that's the case, it can be created by the factory and handed back to the client.

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

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-styled pizzas.

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

Abstract Factory

- So, we delegated the task of creating an object to another object.
- This is called the **abstract factory**.
- As another approach, we may let the subclasses decide.
- The pattern we will see next is called the **factory method**.

Factory Method

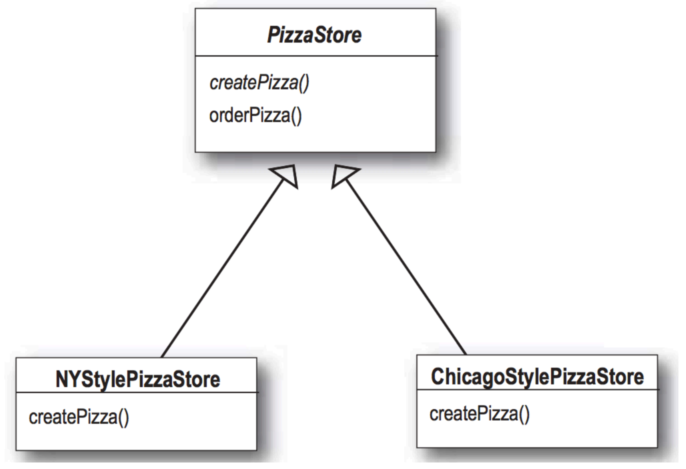
```
public abstract class PizzaStore {


      public Pizza orderPizza(String type) {
            Pizza pizza;

            pizza = createPizza(type);

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

            return pizza;
      }

      abstract createPizza(String type);
}
```

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

All this looks just the same...

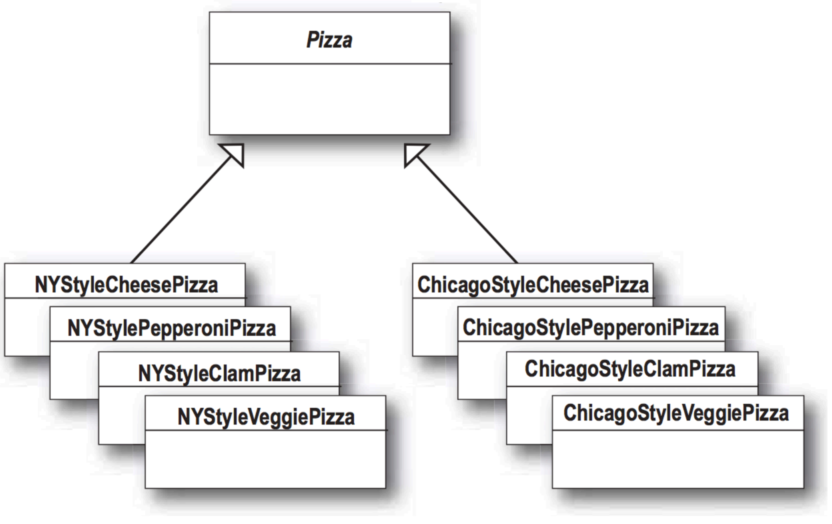
Now we've moved our factory object to this method.

Our "factory method" is now abstract in PizzaStore.

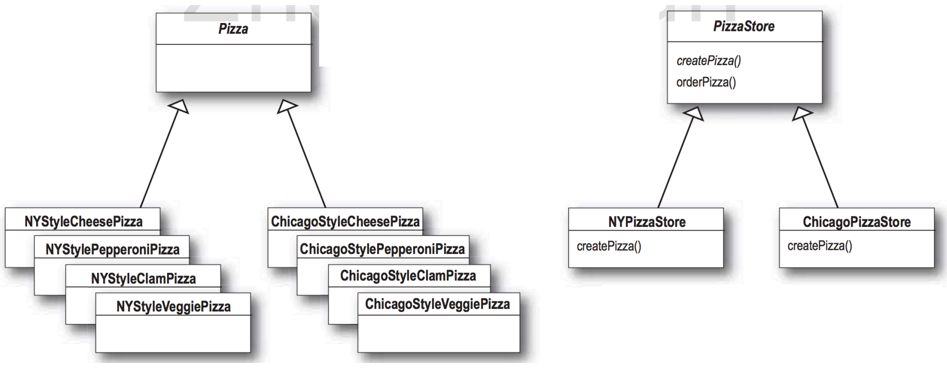


```
public class NYPizzaStore extends PizzaStore {
    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else if (item.equals("clam")) {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

* Note that the orderPizza() method in the superclass has no clue which Pizza we are creating; it just knows it can prepare, bake, cut, and box it!



Parallel Class Hierarchies



```java
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;
    }
}
```

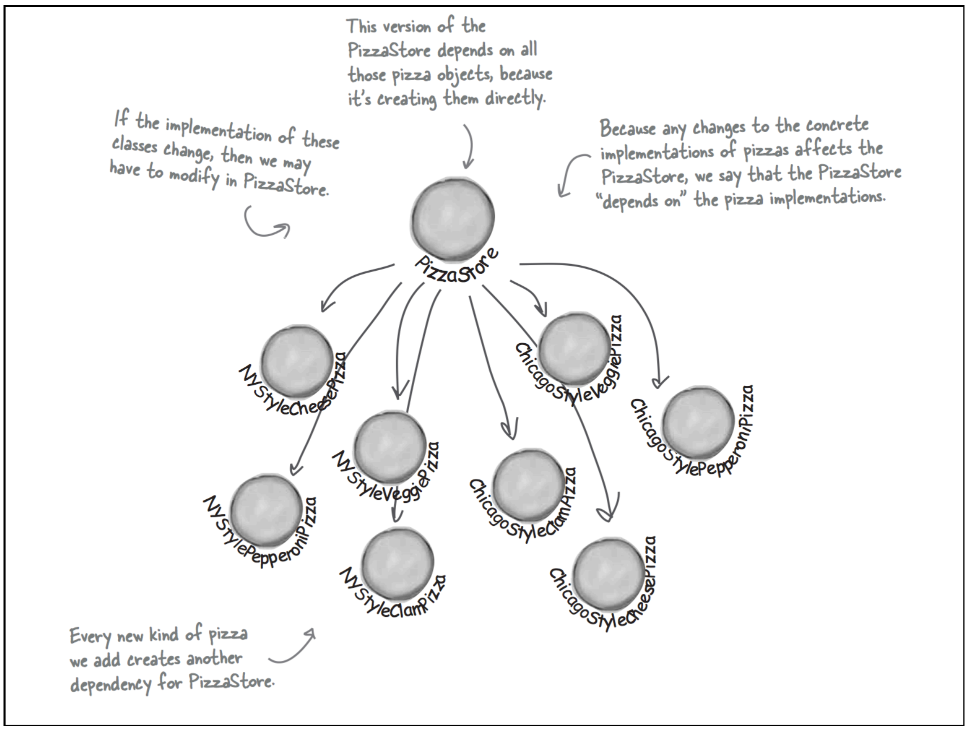
Handles all the NY style pizzas

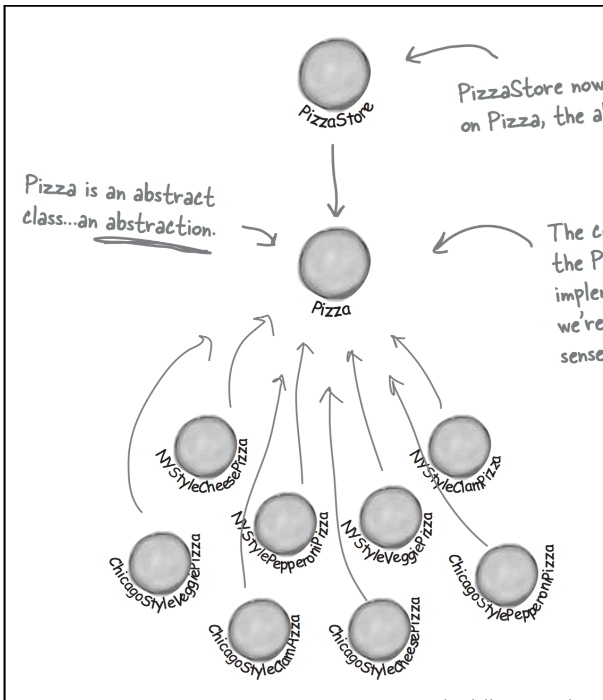
Handles all the Chicago style pizzas

Without the creational patterns.

Design Principle

- Depend upon abstractions. Do not depend upon concrete classes.


This version of the PizzaStore depends on all those pizza objects, because it's creating them directly.
If the implementation of these classes change, then we may have to modify in PizzaStore.
Because any changes to the concrete implementations of pizzas affects the PizzaStore, we say that the PizzaStore "depends on" the pizza implementations.
PizzaStore
NYStyleCheesePizza
ChicagoStyleVeggiePizza
ChicagoStylePepperoniPizza
NYStyleVeggiePizza
NYStylePepperoniPizza
ChicagoStyleClamPizza
NYStyleClamPizza
ChicagoStyleCheesePizza
Every new kind of pizza we add creates another dependency for PizzaStore.


PizzaStore now depends only on Pizza, the abstract class.
PizzaStore
Pizza is an abstract class...an abstraction.
Pizza
The concrete pizza classes depend on the Pizza abstraction too, because they implement the Pizza interface (remember, we're using "interface" in the general sense) in the Pizza abstract class.
NYStyleCheesePizza
NYStyleClamPizza
NYStylePepperoniPizza
NYStyleVeggiePizza
ChicagoStyleVeggiePizza
ChicagoStylePepperoniPizza
ChicagoStyleClamPizza
ChicagoStyleCheesePizza

Dependency Inversion Principle