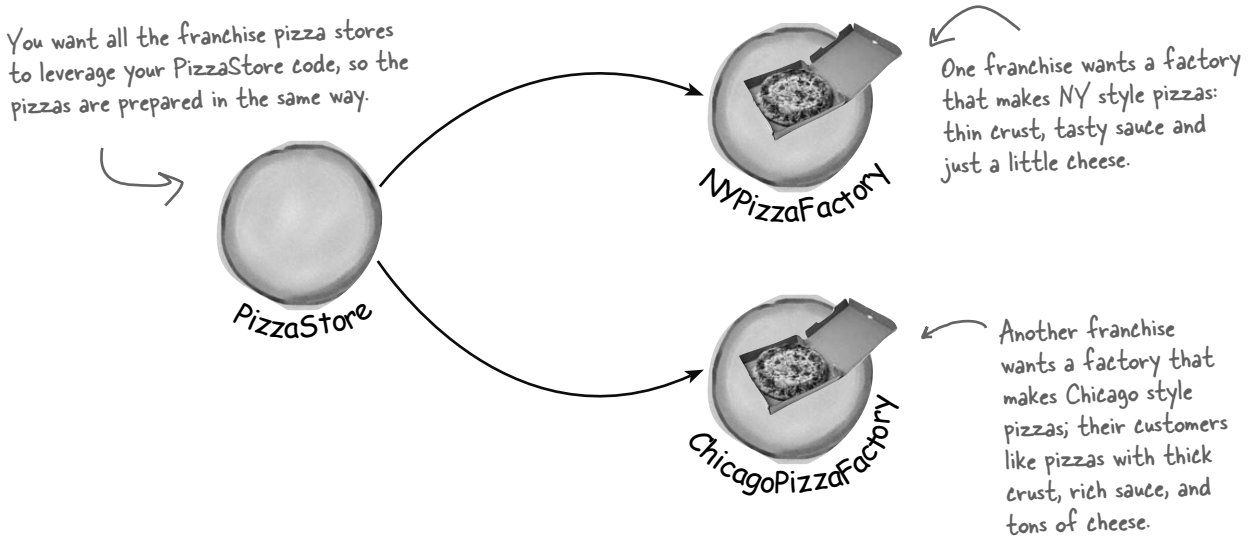
Franchising the pizza store

Your Objectville PizzaStore has done so well that you've trounced the competition and now everyone wants a PizzaStore in their own neighborhood. As the franchiser, you want to ensure the quality of the franchise operations and so you want them to use your time-tested code.

But what about regional differences? Each franchise might want to offer different styles of pizzas (New York, Chicago, and California, to name a few), depending on where the franchise store is located and the tastes of the local pizza connoisseurs.



We've seen one approach...

If we take out SimplePizzaFactory and create three different factories, NYPizzaFactory, ChicagoPizzaFactory and CaliforniaPizzaFactory, then we can just compose the PizzaStore with the appropriate factory and a franchise is good to go. That's one approach.

Let's see what that would look like...

```
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");
```

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 flavored ones

But you'd like a little more quality control...

So you test marketed the SimpleFactory idea, and what you found was that the franchises were using your factory to create pizzas, but starting to employ their own home grown procedures for the rest of the process: they'd bake things a little differently, they'd forget to cut the pizza and they'd use third-party boxes.

Rethinking the problem a bit, you see that what you'd really like to do is create a framework that ties the store and the pizza creation together, yet still allows things to remain flexible.

In our early code, before the SimplePizzaFactory, we had the pizza-making code tied to the PizzaStore, but it wasn't flexible. So, how can we have our pizza and eat it too?

Not what you want in a good franchise. You do NOT want to know what he puts on his pizzas.

A framework for the pizza store

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.

What we're going to do is put the createPizza() method back into PizzaStore, but this time as an **abstract method**, and then create a PizzaStore subclass for each regional style.

First, let's look at the changes to the PizzaStore:

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);
}
```

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.

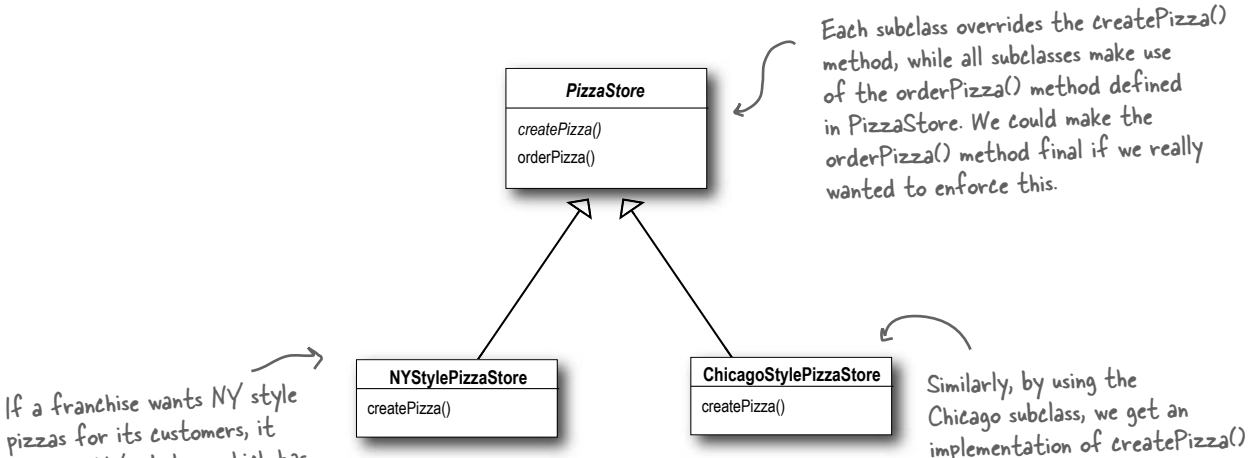
Our "factory method" is now abstract in PizzaStore.

Now we've got a store waiting for subclasses; 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. Let's take a look at how this is going to work.

Allowing the subclasses to decide

Remember, the PizzaStore already has a well-honed order system in the orderPizza() method and you want to ensure that it's consistent across all franchises.

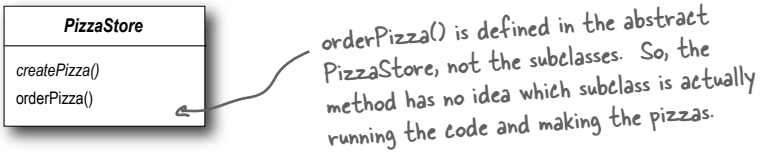
What varies among the regional PizzaStores is the style of pizzas they make – New York Pizza has thin crust, Chicago Pizza has thick, and so on – and we are going to push all these variations into the createPizza() method and make it responsible for creating the right kind of pizza. The way we do this is by letting each subclass of PizzaStore define what the createPizza() method looks like. So, we will have a number of concrete subclasses of PizzaStore, each with its own pizza variations, all fitting within the PizzaStore framework and still making use of the well-tuned orderPizza() method.



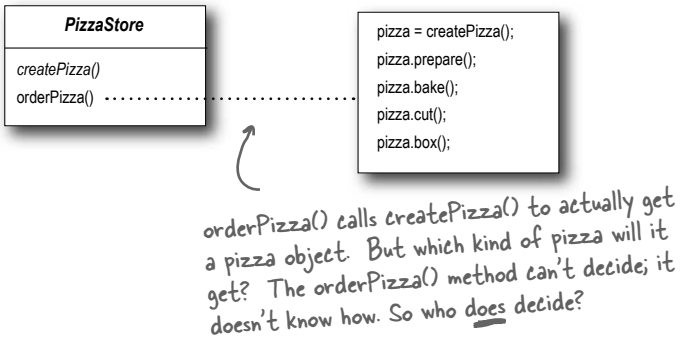

```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new NYStyleCheesePizza();
    } else if (type.equals("pepperoni") {
        pizza = new NYStylePepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new NYStyleClamPizza();
    } else if (type.equals("veggie")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

```
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();
    }
}
```

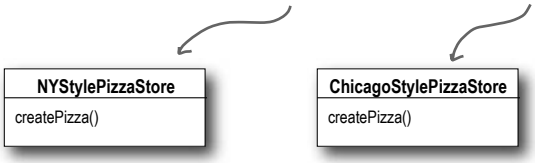
Well, think about it from the point of view of the PizzaStore's orderPizza() method: it is defined in the abstract PizzaStore, but concrete types are only created in the subclasses.



Now, to take this a little further, the orderPizza() method does a lot of things with a Pizza object (like prepare, bake, cut, box), but because Pizza is abstract, orderPizza() has no idea what real concrete classes are involved. In other words, it's decoupled!



When orderPizza() calls createPizza(), one of your subclasses will be called into action to create a pizza. Which kind of pizza will be made? Well, that's decided by the choice of pizza store you order from, NYStylePizzaStore or ChicagoStylePizzaStore.



So, is there a real-time decision that subclasses make? No, but from the perspective of orderPizza(), if you chose a NYStylePizzaStore, that subclass gets to determine which pizza is made. So the subclasses aren't really "deciding" – it was *you* who decided by choosing which store you wanted – but they do determine which kind of pizza gets made.

Let's make a PizzaStore

Being a franchise has its benefits. You get all the PizzaStore functionality for free. All the regional stores need to do is subclass PizzaStore and supply a createPizza() method that implements their style of Pizza. We'll take care of the big three pizza styles for the franchisees.

Here's the New York regional style:

createPizza() returns a Pizza, and the subclass is fully responsible for which concrete Pizza it instantiates

The NYPizzaStore extends PizzaStore, so it inherits the orderPizza() method (among others).

```
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;
    }
}
```

We've got to implement createPizza(), since it is abstract in PizzaStore.

Here's where we create our concrete classes. For each type of Pizza we create the NY style.

* 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!

Once we've got our PizzaStore subclasses built, it will be time to see about ordering up a pizza or two. But before we do that, why don't you take a crack at building the Chicago Style and California Style pizza stores on the next page.

Sharpen your pencil

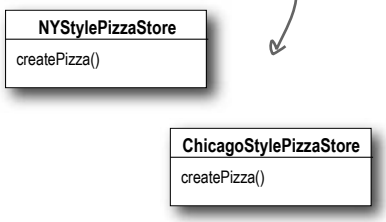
We've knocked out the NYPizzaStore, just two more to go and we'll be ready to franchise! Write the Chicago and California PizzaStore implementations here:

Declaring a factory method

With just a couple of transformations to the PizzaStore we've gone from having an object handle the instantiation of our concrete classes to a set of subclasses that are now taking on that responsibility. Let's take a closer look:

```
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.

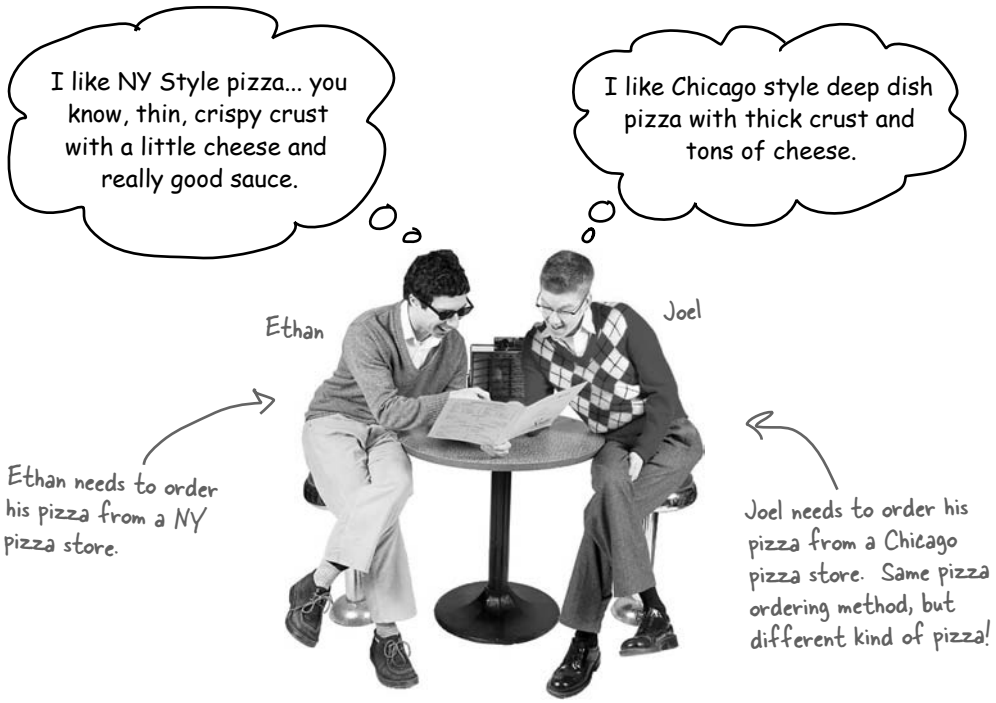
```
abstract Product factoryMethod(String type)
```

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.

Let's see how it works: ordering pizzas with the pizza factory method



So how do they order?

1. First, Joel and Ethan need an instance of a PizzaStore. Joel needs to instantiate a ChicagoPizzaStore and Ethan needs a NYPizzaStore.
2. With a PizzaStore in hand, both Ethan and Joel call the orderPizza() method and pass in the type of pizza they want (cheese, veggie, and so on).
3. To create the pizzas, the createPizza() method is called, which is defined in the two subclasses NYPizzaStore and ChicagoPizzaStore. As we defined them, the NYPizzaStore instantiates a NY style pizza, and the ChicagoPizzaStore instantiates Chicago style pizza. In either case, the Pizza is returned to the orderPizza() method.
4. The orderPizza() method has no idea what kind of pizza was created, but it knows it is a pizza and it prepares, bakes, cuts, and boxes it for Ethan and Joel.

Let's check out how these pizzas are really made to order...

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");
```

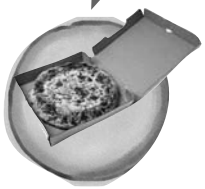
The orderPizza() method is called on the nyPizzaStore instance (the method defined inside PizzaStore runs).

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.

We're just missing one thing: PIZZA!

Our PizzaStore isn't going to be very popular without some pizzas, so let's implement them:

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.

```
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");
    }

    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;
    }
}
```

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

Preparation follows a number of steps in a particular sequence.

REMEMBER: we don't provide import and package statements in the code listings. Get the complete source code from the headfirstlabs web site. You'll find the URL on page xxxiii in the Intro.

Now we just need some concrete subclasses... how about defining New York and Chicago style cheese pizzas?

```
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");
    }
}
```

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

And one topping, 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 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.

You've waited long enough, time for some pizzas!

```
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.

And the other for Joel's.

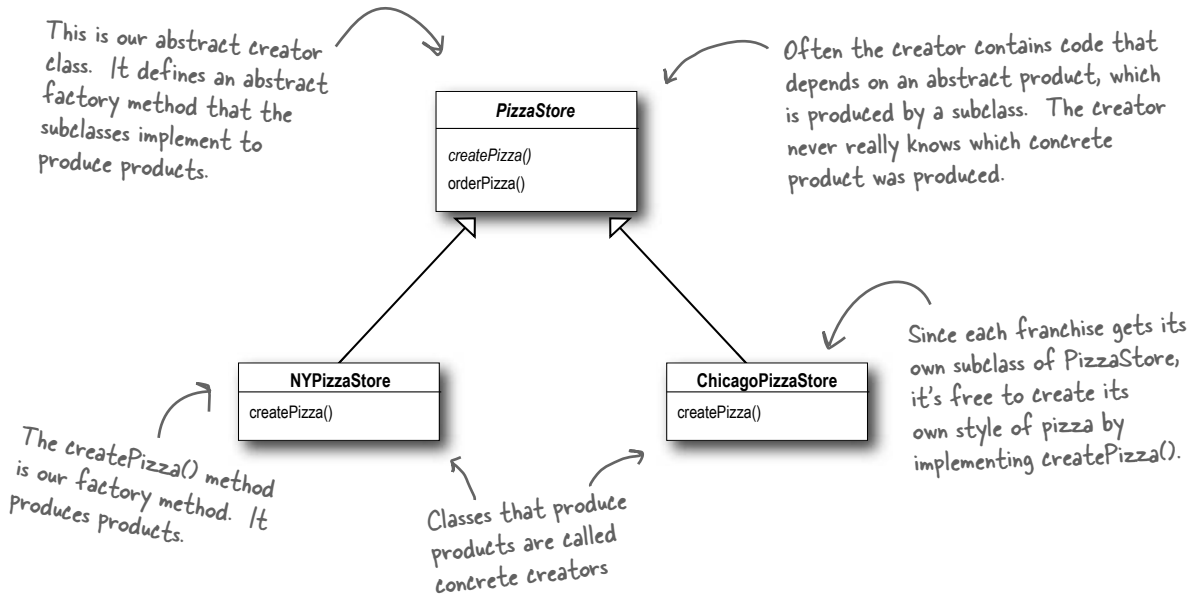
```
File  Edit  Window  Help  YouWantMootzOnThatPizza?
%java PizzaTestDrive

Preparing NY Style Sauce and Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Grated Regiano cheese
Bake for 25 minutes at 350
Cutting the pizza into diagonal slices
Place pizza in official PizzaStore box
Ethan ordered a NY Style Sauce and Cheese Pizza

Preparing Chicago Style Deep Dish Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
   Shredded Mozzarella Cheese
Bake for 25 minutes at 350
Cutting the pizza into square slices
Place pizza in official PizzaStore box
Joel ordered a Chicago Style Deep Dish Cheese Pizza
```

Both pizzas get prepared, the toppings added, and the pizzas baked, cut and boxed. Our superclass never had to know the details, the subclass handled all that just by instantiating the right pizza.

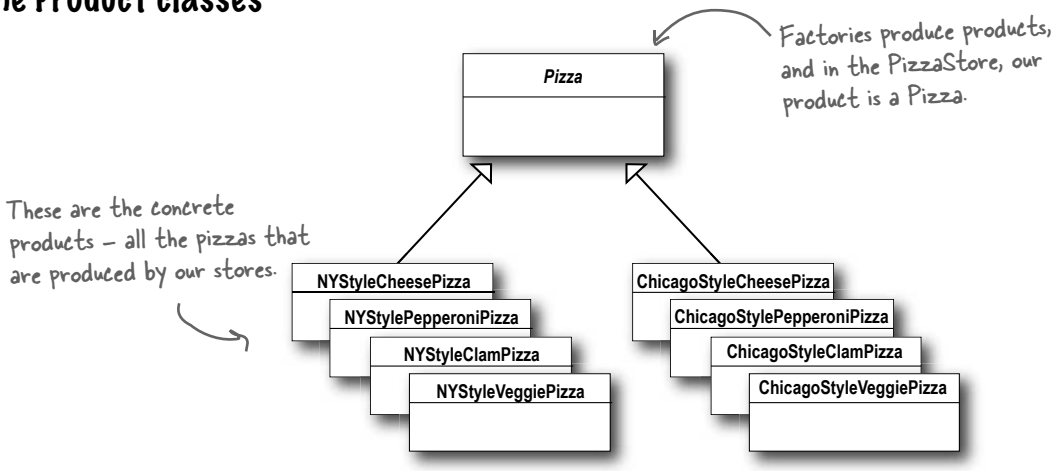
It's finally time to meet the Factory Method Pattern

All factory patterns encapsulate object creation. The Factory Method Pattern encapsulates object creation by letting subclasses decide what objects to create. Let's check out these class diagrams to see who the players are in this pattern:

The Creator classes



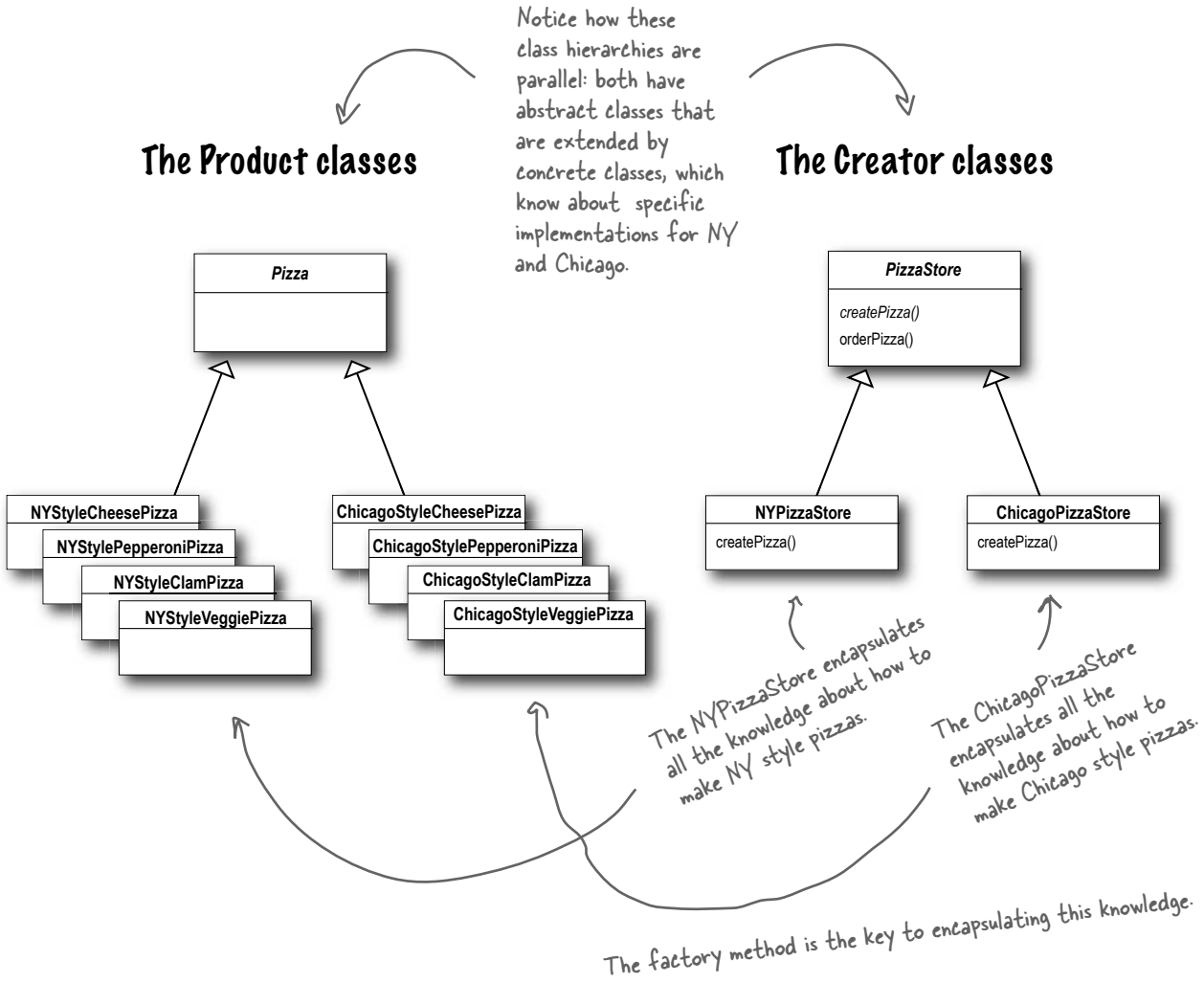
The Product classes



Another perspective: parallel class hierarchies

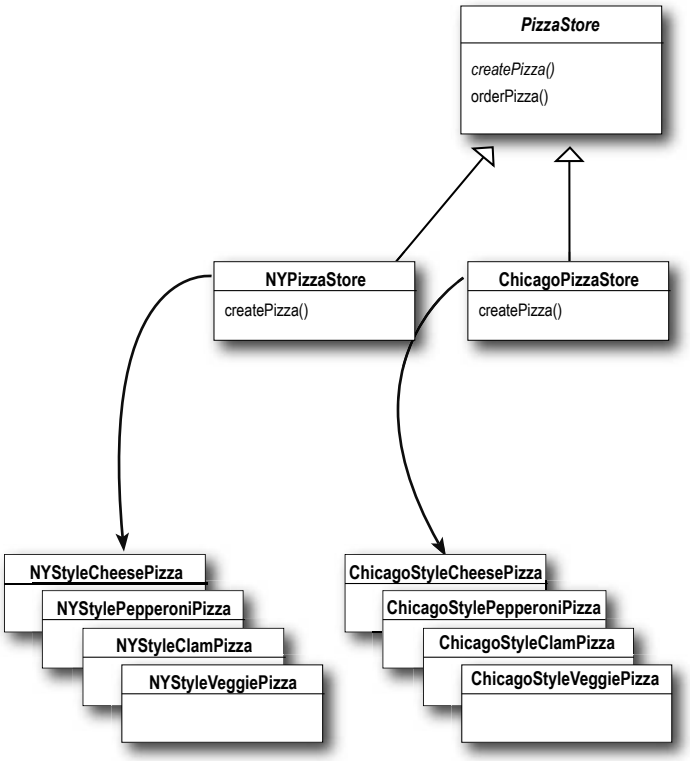
We've seen that the factory method provides a framework by supplying an orderPizza() method that is combined with a factory method. Another way to look at this pattern as a framework is in the way it encapsulates product knowledge into each creator.

Let's look at the two parallel class hierarchies and see how they relate:



Design Puzzle

We need another kind of pizza for those crazy Californians (crazy in a *good* way of course). Draw another parallel set of classes that you'd need to add a new California region to our PizzaStore.



Your drawing here...

Okay, now write the five *most bizarre* things you can think of to put on a pizza. Then, you'll be ready to go into business making pizza in California!

Factory Method Pattern defined

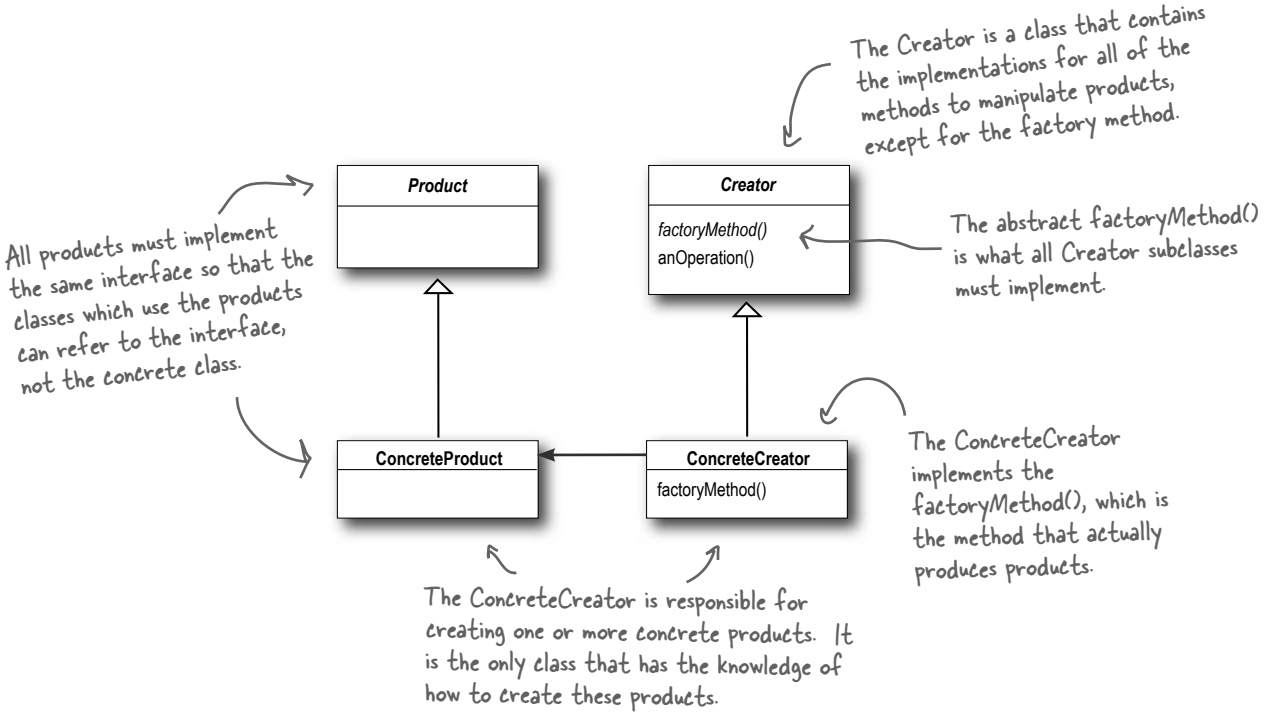
It's time to roll out the official definition of the Factory Method Pattern:

> **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.

As with every factory, the Factory Method Pattern gives us a way to encapsulate the instantiations of concrete types. Looking at the class diagram below, you can see that the abstract Creator gives you an interface with a method for creating objects, also known as the "factory method." Any other methods implemented in the abstract Creator are written to operate on products produced by the factory method. Only subclasses actually implement the factory method and create products.

As in the official definition, you'll often hear developers say that the Factory Method lets subclasses decide which class to instantiate. They say "decides" not because the pattern allows subclasses themselves to decide at runtime, but because the creator class is written without knowledge of the actual products that will be created, which is decided purely by the choice of the subclass that is used.

You could ask them what "decides" means, but we bet you now understand this better than they do!



there are no Dumb Questions

Q: What's the advantage of the Factory Method Pattern when you only have one ConcreteCreator?

A: The Factory Method Pattern is useful if you've only got one concrete creator because you are decoupling the implementation of the product from its use. If you add additional products or change a product's implementation, it will not affect your Creator (because the Creator is not tightly coupled to any ConcreteProduct).

Q: Would it be correct to say that our NY and Chicago stores are implemented using Simple Factory? They look just like it.

A: They're similar, but used in different ways. Even though the implementation of each concrete store looks a lot like the SimplePizzaFactory, remember that the concrete stores are extending a class which has defined createPizza() as an abstract method. It is up to each store to define the behavior of the createPizza() method. In Simple Factory, the factory is another object that is composed with the PizzaStore.

Q: Are the factory method and the Creator always abstract?

A: No, you can define a default factory method to produce some concrete product. Then you always have a means of creating products even if there are no subclasses of the Creator.

Q: Each store can make four different kinds of pizzas based on the type passed in. Do all concrete creators make multiple products, or do they sometimes just make one?

A: We implemented what is known as the parameterized factory method. It can make more than one object based on a parameter passed in, as you noticed. Often, however, a factory just produces one object and is not parameterized. Both are valid forms of the pattern.

Q: Your parameterized types don't seem "type-safe." I'm just passing in a String! What if I asked for a "CalmPizza"?

A: You are certainly correct and that would cause, what we call in the business, a "runtime error." There are several other more sophisticated techniques that can be used to make parameters more "type safe", or, in other words, to ensure errors in parameters can be caught at compile time. For instance, you can create objects that represent the parameter types, use static constants, or, in Java 5, you can use *enums*.

Q: I'm still a bit confused about the difference between Simple Factory and Factory Method. They look very similar, except that in Factory Method, the class that returns the pizza is a subclass. Can you explain?

A: You're right that the subclasses do look a lot like Simple Factory, however think of Simple Factory as a one shot deal, while with Factory Method you are creating a framework that let's the subclasses decide which implementation will be used. For example, the orderPizza() method in the Factory Method provides a general framework for creating pizzas that relies on a factory method to actually create the concrete classes that go into making a pizza. By subclassing the PizzaStore class, you decide what concrete products go into making the pizza that orderPizza() returns. Compare that with SimpleFactory, which gives you a way to encapsulate object creation, but doesn't give you the flexibility of the Factory Method because there is no way to vary the products you're creating.

Master and Student...

Master: *Grasshopper, tell me how your training is going?*

Student: *Master, I have taken my study of "encapsulate what varies" further.*

Master: *Go on...*

Student: *I have learned that one can encapsulate the code that creates objects. When you have code that instantiates concrete classes, this is an area of frequent change. I've learned a technique called "factories" that allows you to encapsulate this behavior of instantiation.*

Master: *And these "factories," of what benefit are they?*

Student: *There are many. 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.*

Master: *Yes Grasshopper, your OO instincts are growing. Do you have any questions for your master today?*

Student: *Master, I know that by encapsulating object creation I am coding to abstractions and decoupling my client code from actual implementations. But my factory code must still use concrete classes to instantiate real objects. Am I not pulling the wool over my own eyes?*

Master: *Grasshopper, object creation is a reality of life; we must create objects or we will never create a single Java program. But, with knowledge of this reality, we can design our code so that we have corralled this creation code like the sheep whose wool you would pull over your eyes. Once corralled, we can protect and care for the creation code. If we let our creation code run wild, then we will never collect its "wool."*

Student: *Master, I see the truth in this.*

Master: *As I knew you would. Now, please go and meditate on object dependencies.*