

Chapter 2. The Observer Pattern: Keeping your Objects in the know



Don't miss out when something interesting happens! We've got a pattern that keeps your objects in the know when something they might care about happens. Objects can even decide at runtime whether they want to be kept informed. The Observer Pattern is one of the most heavily used patterns in the JDK, and it's incredibly useful. Before we're done, we'll also look at one-to-many relationships and loose coupling (yeah, that's right, we said coupling). With Observer, you'll be the life of the Patterns Party.

Congratulations!

Your team has just won the contract to build Weather-O-Rama, Inc.'s next-generation, Internet-based Weather Monitoring Station.



Statement of Work

Congratulations on being selected to build our next-generation, Internet-based Weather Monitoring Station!

The weather station will be based on our patent pending WeatherData object, which tracks current weather conditions (temperature, humidity, and barometric pressure). We'd like you to create an application that initially provides three display elements: current conditions, weather statistics, and a simple forecast, all updated in real time as the WeatherData object acquires the most recent measurements.

Further, this is an expandable weather station. Weather-ORama wants to release an API so that other developers can write their own weather displays and plug them right in. We'd like for you to supply that API!

Weather-O-Rama thinks we have a great business model: once the customers are hooked, we intend to charge them for each display they use. Now for the best part: we are going to pay you in stock options.

We look forward to seeing your design and alpha application.

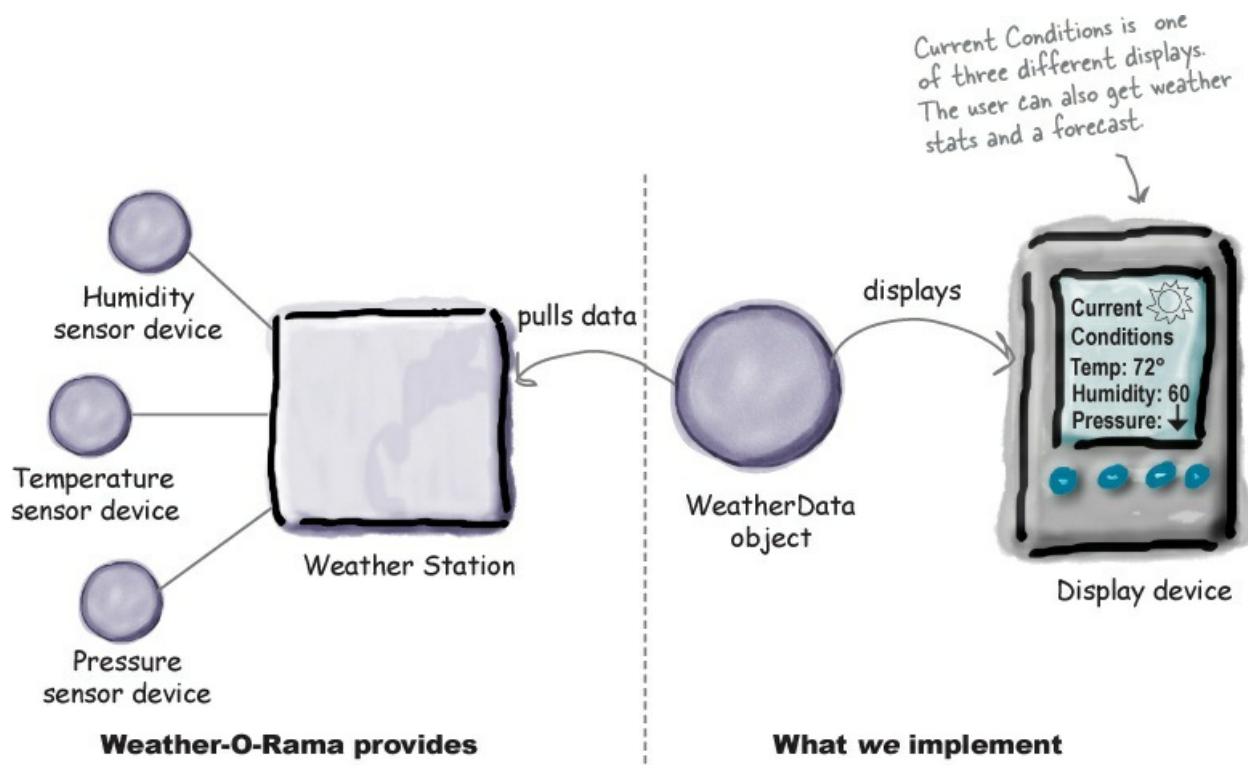
Sincerely,

Johnny Hurricane, CEO

P.S. We are overnighting the WeatherData source files to you.

The Weather Monitoring application overview

The three players in the system are the weather station (the physical device that acquires the actual weather data), the WeatherData object (that tracks the data coming from the Weather Station and updates the displays), and the display that shows users the current weather conditions.



The WeatherData object knows how to talk to the physical Weather Station, to get updated data. The WeatherData object then updates its displays for the three different display elements: Current Conditions (shows temperature, humidity, and pressure), Weather Statistics, and a simple forecast.

Our job, if we choose to accept it, is to create an app that uses the WeatherData object to update three displays for current conditions, weather stats, and a forecast.

Unpacking the WeatherData class

As promised, the next morning the WeatherData source files arrive. When we peek inside the code, things look pretty straightforward:

```

WeatherData
getTemperature()
getHumidity()
getPressure()
measurementsChanged()
// other methods

```

These three methods return the most recent weather measurements for temperature, humidity, and barometric pressure, respectively.

We don't care HOW these variables are set; the WeatherData object knows how to get updated info from the Weather Station.

The developers of the WeatherData object left us a clue about what we need to add...

```

/*
 * This method gets called
 * whenever the weather measurements
 * have been updated
 *
 */
public void measurementsChanged() {
    // Your code goes here
}

```

WeatherData.java

Remember, this Current Conditions is just ONE of three different display screens. ↓



Display device

Our job is to implement measurementsChanged() so that it updates the three displays for current conditions, weather stats, and forecast.

What do we know so far?



The spec from Weather-O-Rama wasn't all that clear, but we have to figure out what we need to do. So, what do we know so far?

<input checked="" type="checkbox"/>	The WeatherData class has getter methods for three measurement values: temperature, humidity, and barometric pressure. <code>getTemperature()</code> <code>getHumidity()</code> <code>getPressure()</code>
<input checked="" type="checkbox"/>	The measurementsChanged() method is called any time new weather measurement data is available. (We don't know or care how this method is called; we just know that it <i>is</i> .) <code>measurementsChanged()</code>
<input checked="" type="checkbox"/>	We need to implement three display elements that use the weather data: a <i>current conditions</i> display, a <i>statistics display</i> , and a <i>forecast</i> display. These displays must be updated each time WeatherData has new measurements. ☰
<input checked="" type="checkbox"/>	The system must be expandable — other developers can create new custom display elements and users can add or remove as many display elements as they want to the application. Currently, we know about only the initial <i>three</i> display types (current conditions, statistics, and forecast). ☰

Taking a first, misguided SWAG at the Weather Station

Here's a first implementation possibility — we'll take the hint from the Weather-O-Rama developers and add our code to the `measurementsChanged()` method:

```

public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {
        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();
    }

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
}

// other WeatherData methods here
}

```

Grab the most recent measurements by calling the WeatherData's getter methods (already implemented).

Now update the displays...

Call each display element to update its display, passing it the most recent measurements.

SHARPEN YOUR PENCIL

Based on our first implementation, which of the following apply? (Choose all that apply.)

<input type="checkbox"/>	A. We are coding to concrete implementations, not interfaces.
<input type="checkbox"/>	B. For every new display element we need to alter code.
<input type="checkbox"/>	C. We have no way to add (or remove) display elements at run time.
<input type="checkbox"/>	D. The display elements don't implement a common interface.
<input type="checkbox"/>	E. We haven't encapsulated the part that changes.
<input type="checkbox"/>	F. We are violating encapsulation of the WeatherData class.

Definition of SWAG: Scientific Wild A** Guess

What's wrong with our implementation?

Think back to all those Chapter 1 concepts and principles...

```

public void measurementsChanged() {

    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
}

```

Area of change. We need to encapsulate this.

By coding to concrete implementations we have no way to add or remove other display elements without making changes to the program.

At least we seem to be using a common interface to talk to the display elements... they all have an update() method that takes the temp, humidity, and pressure values.



We'll take a look at Observer, then come back and figure out how to apply it to the Weather Monitoring app.

Meet the Observer Pattern

You know how newspaper or magazine subscriptions work:

- ① A newspaper publisher goes into business and begins publishing

newspapers.

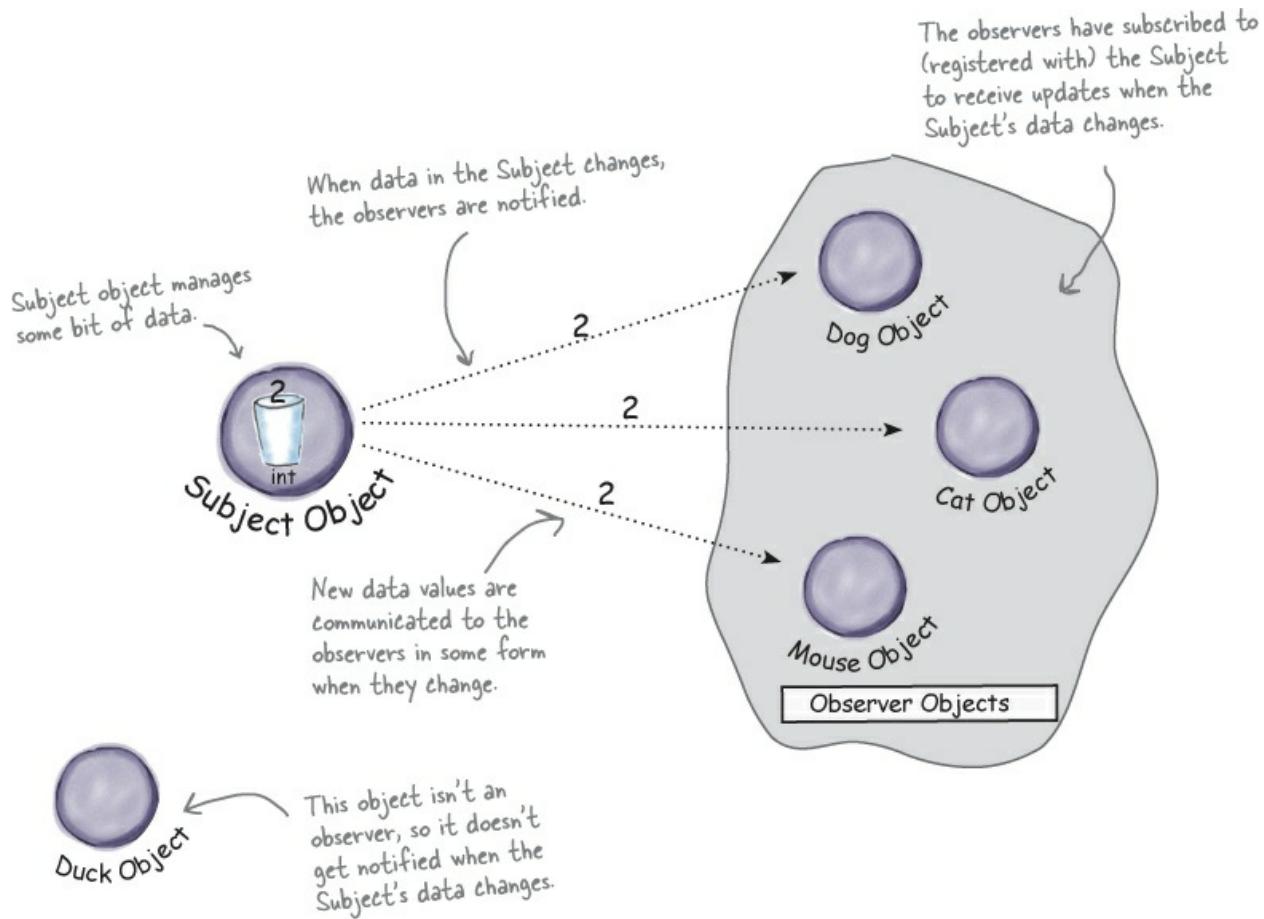
- ② You subscribe to a particular publisher, and every time there's a new edition it gets delivered to you. As long as you remain a subscriber, you get new newspapers.
- ③ You unsubscribe when you don't want papers anymore, and they stop being delivered.
- ④ While the publisher remains in business, people, hotels, airlines, and other businesses constantly subscribe and unsubscribe to the newspaper.



Publishers + Subscribers = Observer Pattern

If you understand newspaper subscriptions, you pretty much understand the Observer Pattern, only we call the publisher the SUBJECT and the subscribers the OBSERVERS.

Let's take a closer look:



A day in the life of the Observer Pattern

A Duck object comes along and tells the Subject that it wants to become an observer.	
---	--

Duck really wants in on the action; those ints Subject is sending out whenever its state changes look pretty interesting...

The Duck object is now an official observer.	
---	--

Duck is psyched... he's on the list and is waiting with great anticipation for the next notification so he can get an int.

The Subject gets a new data value!	
---	--

Now Duck and all the rest of the observers get a notification that the Subject has changed.

The Mouse object asks to be removed as an observer.	
--	--

The Mouse object has been getting ints for ages and is tired of it, so it decides it's

time to stop being an observer.

Mouse is outta here!

The Subject acknowledges the Mouse's request and removes it from the set of observers.



The Subject has another new int.

All the observers get another notification, except for the Mouse who is no longer included. Don't tell anyone, but the Mouse secretly misses those ints... maybe it'll ask to be an observer again some day.



Five-minute drama: a subject for observation



In today's skit, two post-bubble software developers encounter a real live head hunter...



1



2

Headhunter/Subject

Software Developer #1



3

Software Developer #2



4

Subject

- 5** Meanwhile, for Lori and Jill life goes on; if a Java job comes along, they'll get notified. After all, they are observers.



6

Subject



Jill lands her own job!



8

Observer



Two weeks later...



Jill's loving life, and no longer an observer. She's also enjoying the nice fat signing bonus that she got because the company didn't have to pay a headhunter.



But what has become of our dear Lori? We hear she's beating the headhunter at his own game. She's not only still an observer, she's got her own call list now, and she is notifying her own observers. Lori's a subject and an observer all in one.

The Observer Pattern defined

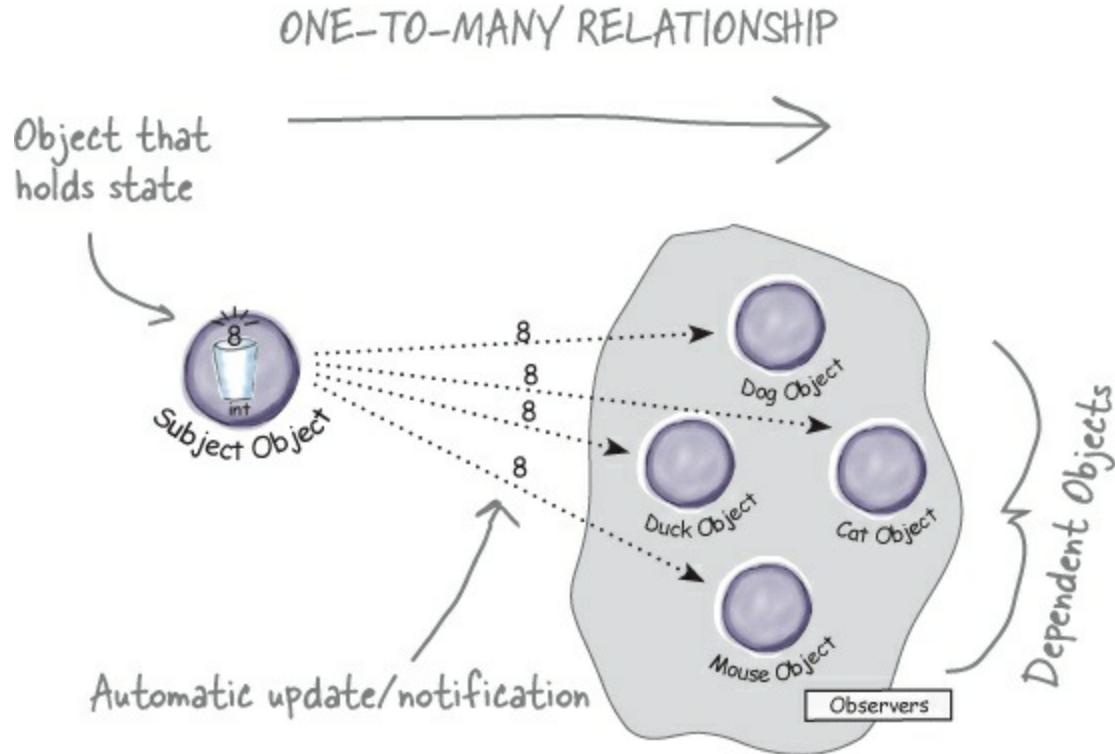
When you're trying to picture the Observer Pattern, a newspaper subscription service with its publisher and subscribers is a good way to visualize the pattern.

In the real world, however, you'll typically see the Observer Pattern defined like this:

NOTE

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

Let's relate this definition to how we've been talking about the pattern:



The Observer Pattern defines a one-to-many relationship between a set of objects. When the state of one object changes, all of its dependents are notified.

The subject and observers define the one-to-many relationship. The observers are dependent on the subject such that when the subject's state changes, the observers get notified. Depending on the style of notification, the observer may also be updated with new values.

As you'll discover, there are a few different ways to implement the Observer Pattern, but most revolve around a class design that includes Subject and Observer interfaces.

Let's take a look...

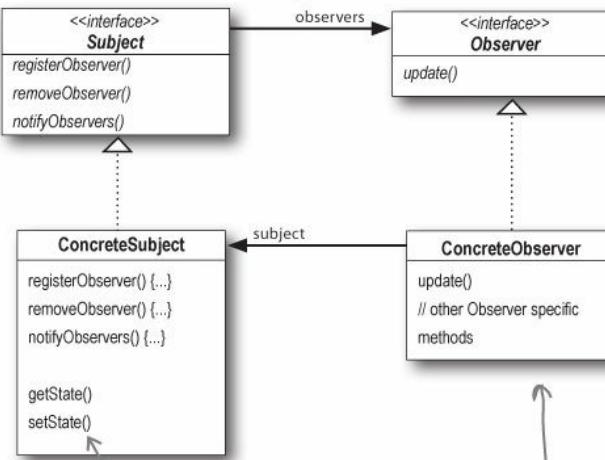
The Observer Pattern defined: the class diagram

Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from being observers.

Each subject can have many observers.

All potential observers need to implement the Observer interface. This interface just has one method, update(), that gets called when the Subject's state changes.

A concrete subject always implements the Subject interface. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method that is used to update all the current observers whenever state changes.



The concrete subject may also have methods for setting and getting its state (more about this later).

Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

THERE ARE NO DUMB QUESTIONS

Q: Q: What does this have to do with one-to-many relationships?

A: A: With the Observer Pattern, the Subject is the object that contains the state and controls it. So, there is ONE subject with state. The observers, on the other hand, use the state, even if they don't own it. There are many observers and they rely on the Subject to tell them when its state changes. So there is a relationship between the ONE Subject to the MANY Observers.

Q: Q: How does dependence come into this?

A: A: Because the subject is the sole owner of that data, the observers are dependent on the subject to update them when the data changes. This leads to a cleaner OO design than allowing many objects to control the same data.

The power of Loose Coupling

When two objects are loosely coupled, they can interact, but have very little knowledge of each other.

The Observer Pattern provides an object design where subjects and observers are loosely coupled.

Why?

The only thing the subject knows about an observer is that it implements a certain interface (the Observer interface). It doesn't need to know the concrete class of the observer, what it does, or anything else about it.

We can add new observers at any time. Because the only thing the subject depends on is a list of objects that implement the Observer interface, we can add new observers whenever we want. In fact, we can replace any observer at runtime with another observer and the subject will keep purring along. Likewise, we can remove observers at any time.

We never need to modify the subject to add new types of observers. Let's say we have a new concrete class come along that needs to be an observer. We don't need to make any changes to the subject to accommodate the new class type; all we have to do is implement the Observer interface in the new class and register as an observer. The subject doesn't care; it will deliver notifications to any object that implements the Observer interface.

We can reuse subjects or observers independently of each other. If we have another use for a subject or an observer, we can easily reuse them because the two aren't tightly coupled.

Changes to either the subject or an observer will not affect the other.

Because the two are loosely coupled, we are free to make changes to either, as long as the objects still meet their obligations to implement the subject or observer interfaces.

NOTE

How many different kinds of change can you identify here?

DESIGN PRINCIPLE

Strive for loosely coupled designs between objects that interact.

Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.

SHARPEN YOUR PENCIL

Before moving on, try sketching out the classes you'll need to implement the Weather Station, including the WeatherData class and its display elements. Make sure your diagram shows how all the pieces fit together and also how another developer might implement her own display element.

If you need a little help, read the next page; your teammates are already talking about how to design the Weather Station.

Cubicle conversation

Back to the Weather Station project. Your teammates have already started thinking through the problem...



Mary: Well, it helps to know we're using the Observer Pattern.

Sue: Right... but how do we apply it?

Mary: Hmm. Let's look at the definition again:

The Observer Pattern defines a one-to-many dependency between objects so

that when one object changes state, all its dependents are notified and updated automatically.

Mary: That actually makes some sense when you think about it. Our WeatherData class is the “one” and our “many” is the various display elements that use the weather measurements.

Sue: That’s right. The WeatherData class certainly has state... that’s the temperature, humidity, and barometric pressure, and those definitely change.

Mary: Yup, and when those measurements change, we have to notify all the display elements so they can do whatever it is they are going to do with the measurements.

Sue: Cool, I now think I see how the Observer Pattern can be applied to our Weather Station problem.

Mary: There are still a few things to consider that I’m not sure I understand yet.

Sue: Like what?

Mary: For one thing, how do we get the weather measurements to the display elements?

Sue: Well, looking back at the picture of the Observer Pattern, if we make the WeatherData object the subject, and the display elements the observers, then the displays will register themselves with the WeatherData object in order to get the information they want, right?

Mary: Yes... and once the Weather Station knows about a display element, then it can just call a method to tell it about the measurements.

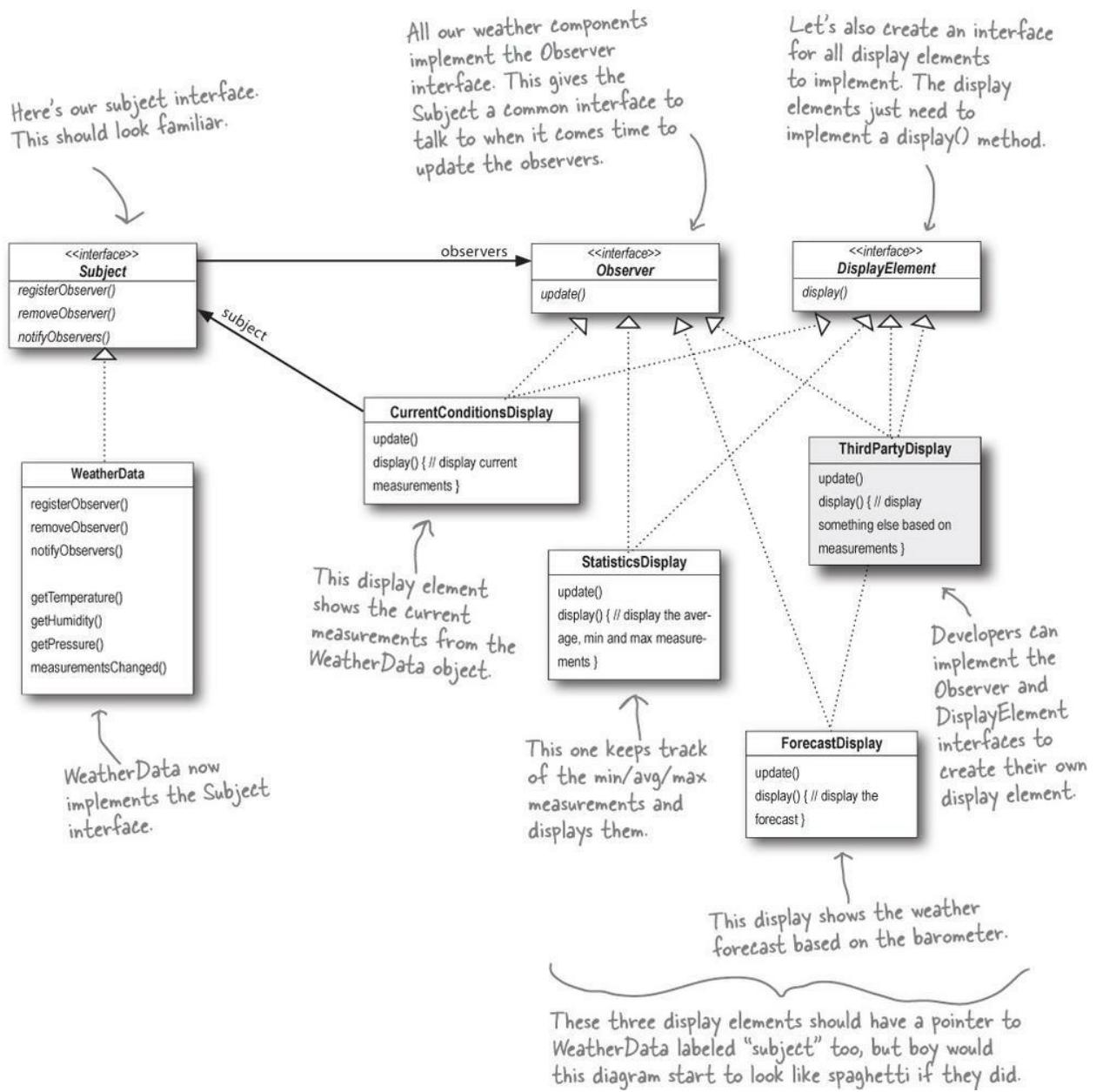
Sue: We gotta remember that every display element can be different... so I think that’s where having a common interface comes in. Even though every component has a different type, they should all implement the same interface so that the WeatherData object will know how to send them the measurements.

Mary: I see what you mean. So every display will have, say, an update() method that WeatherData will call.

Sue: And update() is defined in a common interface that all the elements implement...

Designing the Weather Station

How does this diagram compare with yours?



Implementing the Weather Station

We're going to start our implementation using the class diagram and following Mary and Sue's lead (from a few pages back). You'll see later in this chapter that Java provides some built-in support for the Observer Pattern, however, we're going to get our hands dirty and roll our own for now. While in some cases you can make use of Java's built-in support, in a lot of cases it's more flexible to build your own (and it's not all that hard). So, let's get started with the interfaces:

```

public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}

```

Both of these methods take an Observer as an argument; that is, the Observer to be registered or removed.

This method is called to notify all observers when the Subject's state has changed.

```

public interface Observer {
    public void update(float temp, float humidity, float pressure);
}

```

These are the state values the Observers get from the Subject when a weather measurement changes

```

public interface DisplayElement {
    public void display();
}

```

The DisplayElement interface just includes one method, display(), that we will call when the display element needs to be displayed.

The Observer interface is implemented by all observers, so they all have to implement the update() method. Here we're following Mary and Sue's lead and passing the measurements to the observers.

BRAIN POWER

Mary and Sue thought that passing the measurements directly to the observers was the most straightforward method of updating state. Do you think this is wise? Hint: is this an area of the application that might change in the future? If it did change, would the change be well encapsulated, or would it require changes in many parts of the code?

Can you think of other ways to approach the problem of passing the updated state to the observers?

Don't worry; we'll come back to this design decision after we finish the initial implementation.

Implementing the Subject interface in WeatherData

REMEMBER: we don't provide import and package statements in the code listings.
Get the complete source code from <http://wickedlysmart.com/head-first-design-patterns/>.

Remember our first attempt at implementing the WeatherData class at the beginning of the chapter? You might want to refresh your memory. Now it's time to go back and do things with the Observer Pattern in mind...

```

public class WeatherData implements Subject {
    private ArrayList<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList<Observer>();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    // other WeatherData methods here
}

```

Here we implement the Subject interface.

WeatherData now implements the Subject interface.

We've added an ArrayList to hold the Observers, and we create it in the constructor.

When an observer registers, we just add it to the end of the list.

Likewise, when an observer wants to unregister, we just take it off the list.

Here's the fun part; this is where we tell all the observers about the state. Because they are all Observers, we know they all implement update(), so we know how to notify them.

We notify the Observers when we get updated measurements from the Weather Station.

Okay, while we wanted to ship a nice little weather station with each book, the publisher wouldn't go for it. So, rather than reading actual weather data off a device, we're going to use this method to test our display elements. Or, for fun, you could write code to grab measurements off the Web.

Now, let's build those display elements

Now that we've got our WeatherData class straightened out, it's time to build the Display Elements. Weather-O-Rama ordered three: the current conditions display, the statistics display, and the forecast display. Let's take a look at the current conditions display; once you have a good feel for this display element, check out the statistics and forecast displays in the code directory. You'll see they are very similar.

```

public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display(); ← When update() is called, we
    }                                save the temp and humidity
}                                         and call display().
                                         ↓
                                         The display() method
                                         just prints out the most
                                         recent temp and humidity.

```

This display implements Observer so it can get changes from the WeatherData object.

It also implements DisplayElement, because our API is going to require all display elements to implement this interface.

THERE ARE NO DUMB QUESTIONS

Q: Q: Is update() the best place to call display?

A: A: In this simple example it made sense to call display() when the values changed. However, you are right; there are much better ways to design the way the data gets displayed. We are going to see this when we get to the Model-View-Controller pattern.

Q: Q: Why did you store a reference to the Subject? It doesn't look like you use it again after the constructor.

A: A: True, but in the future we may want to un-register ourselves as an observer and it would be handy to already have a reference to the subject.

Power up the Weather Station



① First, let's create a test harness.

The Weather Station is ready to go. All we need is some code to glue everything together. Here's our first attempt. We'll come back later in the book and make sure all the components are easily pluggable via a configuration file. For now here's how it all works:

```
public class WeatherStation {  
    public static void main(String[] args) {  
        WeatherData weatherData = new WeatherData();  
  
        if you don't  
        want to  
        download the  
        code, you can  
        comment out  
        these two lines  
        and run it.  
        {  
            CurrentConditionsDisplay currentDisplay =  
                new CurrentConditionsDisplay(weatherData);  
            StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);  
            ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);  
  
            weatherData.setMeasurements(80, 65, 30.4f);  
            weatherData.setMeasurements(82, 70, 29.2f);  
            weatherData.setMeasurements(78, 90, 29.2f);  
        }  
    }  
}
```

First, create the WeatherData object.

Create the three displays and pass them the WeatherData object.

Simulate new weather measurements.

② Run the code and let the Observer Pattern do its magic.

```
File Edit Window Help StormyWeather  
%java WeatherStation  
Current conditions: 80.0F degrees and 65.0% humidity  
Avg/Max/Min temperature = 80.0/80.0/80.0  
Forecast: Improving weather on the way!  
Current conditions: 82.0F degrees and 70.0% humidity  
Avg/Max/Min temperature = 81.0/82.0/80.0  
Forecast: Watch out for cooler, rainy weather  
Current conditions: 78.0F degrees and 90.0% humidity  
Avg/Max/Min temperature = 80.0/82.0/78.0  
Forecast: More of the same  
%
```

SHARPEN YOUR PENCIL

Johnny Hurricane, Weather-O-Rama's CEO, just called and they can't possibly ship without a Heat Index display element. Here are the details.

The heat index is an index that combines temperature and humidity to determine the apparent temperature (how hot it actually feels). To compute the heat index, you take the

temperature, T, and the relative humidity, RH, and use this formula:

```
heatindex =  
  
16.923 + 1.85212 * 10-1 * T + 5.37941 * RH - 1.00254 * 10-1 *  
T * RH + 9.41695 * 10-3 * T2 + 7.28898 * 10-3 * RH2 + 3.45372 *  
10-4 * T2 * RH - 8.14971 * 10-4 * T * RH2 + 1.02102 * 10-5 * T2 *  
RH2 - 3.8646 * 10-5 * T3 + 2.91583 * 10-5 * RH3 + 1.42721 * 10-6  
* T3 * RH + 1.97483 * 10-7 * T * RH3 - 2.18429 * 10-8 * T3 * RH2  
+ 8.43296 * 10-10 * T2 * RH3 - 4.81975 * 10-11 * T3 * RH3
```

So get typing!

Just kidding. Don't worry, you won't have to type that formula in; just create your own HeatIndexDisplay.java file and copy the formula from heatindex.txt into it.

NOTE

You can get heatindex.txt from wickedlysmart.com.

How does it work? You'd have to refer to *Head First Meteorology*, or try asking someone at the National Weather Service (or try a web search).

When you finish, your output should look like this:

Here's what
changed in
this output.

```
File Edit Window Help OverdaRainbow  
%java WeatherStation  
Current conditions: 80.0F degrees and 65.0% humidity  
Avg/Max/Min temperature = 80.0/80.0/80.0  
Forecast: Improving weather on the way!  
Heat index is 82.95535  
Current conditions: 82.0F degrees and 70.0% humidity  
Avg/Max/Min temperature = 81.0/82.0/80.0  
Forecast: Watch out for cooler, rainy weather  
Heat index is 86.90124  
Current conditions: 78.0F degrees and 90.0% humidity  
Avg/Max/Min temperature = 80.0/82.0/78.0  
Forecast: More of the same  
Heat index is 83.64967  
%
```

FIRESIDE CHATS

Tonight's talk: **A Subject and Observer spar over the right way to get state information to the Observer.**

Subject:	Observer:
----------	-----------

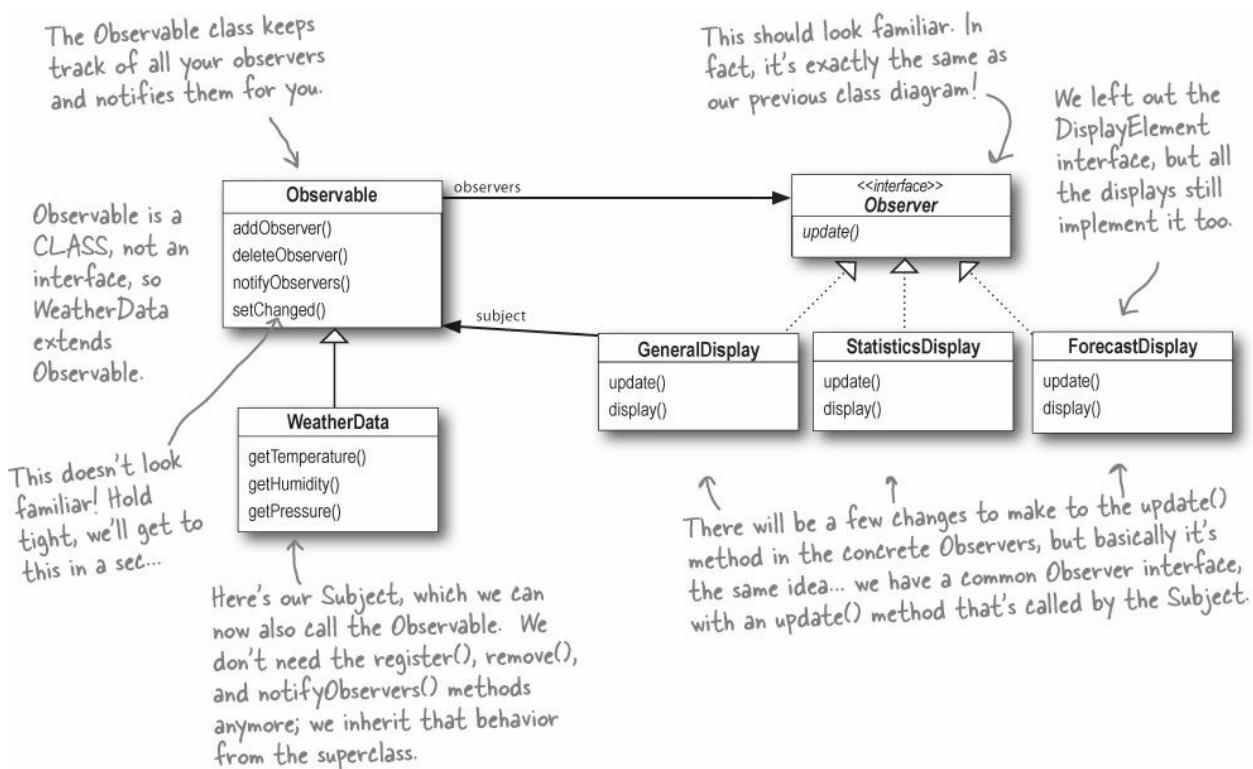
I'm glad we're finally getting a chance to chat in person.	
	Really? I thought you didn't care much about us Observers.
Well, I do my job, don't I? I always tell you what's going on... Just because I don't really know who you are doesn't mean I don't care. And besides, I do know the most important thing about you — you implement the Observer interface.	
	Yeah, but that's just a small part of who I am. Anyway, I know a lot more about you...
Oh yeah, like what?	
	Well, you're always passing your state around to us Observers so we can see what's going on inside you. Which gets a little annoying at times...
Well, excuuuse me. I have to send my state with my notifications so all you lazy Observers will know what happened!	
	Okay, wait just a minute here; first, we're not lazy, we just have other stuff to do in between your oh-so-important notifications, Mr. Subject, and second, why don't you let us come to you for the state we want rather than pushing it out to just everyone?
Well... I guess that might work. I'd have to open myself up even more, though, to let all you Observers come in and get the state that you need. That might be kind of dangerous. I can't let you come in and just snoop around looking at everything I've got.	
	Why don't you just write some public getter methods that will let us pull out the state we need?
Yes, I could let you pull my state. But won't that be less	

convenient for you? If you have to come to me every time you want something, you might have to make multiple method calls to get all the state you want. That's why I like push better... then you have everything you need in one notification.	
	Don't be so pushy! There are so many different kinds of us Observers, there's no way you can anticipate everything we need. Just let us come to you to get the state we need. That way, if some of us only need a little bit of state, we aren't forced to get it all. It also makes things easier to modify later. Say, for example, you expand yourself and add some more state. If you use pull, you don't have to go around and change the update calls on every observer; you just need to change yourself to allow more getter methods to access our additional state.
Well, I can see the advantages to doing it both ways. I have noticed that there is a built-in Java Observer Pattern that allows you to use either push or pull.	
	Oh really? I think we're going to look at that next....
Great... maybe I'll get to see a good example of pull and change my mind.	
	What, us agree on something? I guess there's always hope.

Using Java's built-in Observer Pattern

So far we've rolled our own code for the Observer Pattern, but Java has built-in support in several of its APIs. The most general is the `Observer` interface and the `Observable` class in the `java.util` package. These are quite similar to our `Subject` and `Observer` interfaces, but give you a lot of functionality out of the box. You can also implement either a push or pull style of update to your observers, as you will see.

To get a high-level feel for `java.util.Observer` and `java.util.Observable`, check out this reworked OO design for the `WeatherStation`:



How Java's built-in Observer Pattern works

The built-in Observer Pattern works a bit differently than the implementation

that we used on the Weather Station. The most obvious difference is that WeatherData (our subject) now extends the Observable class and inherits the add, delete, and notify Observer methods (among a few others). Here's how we use Java's version:

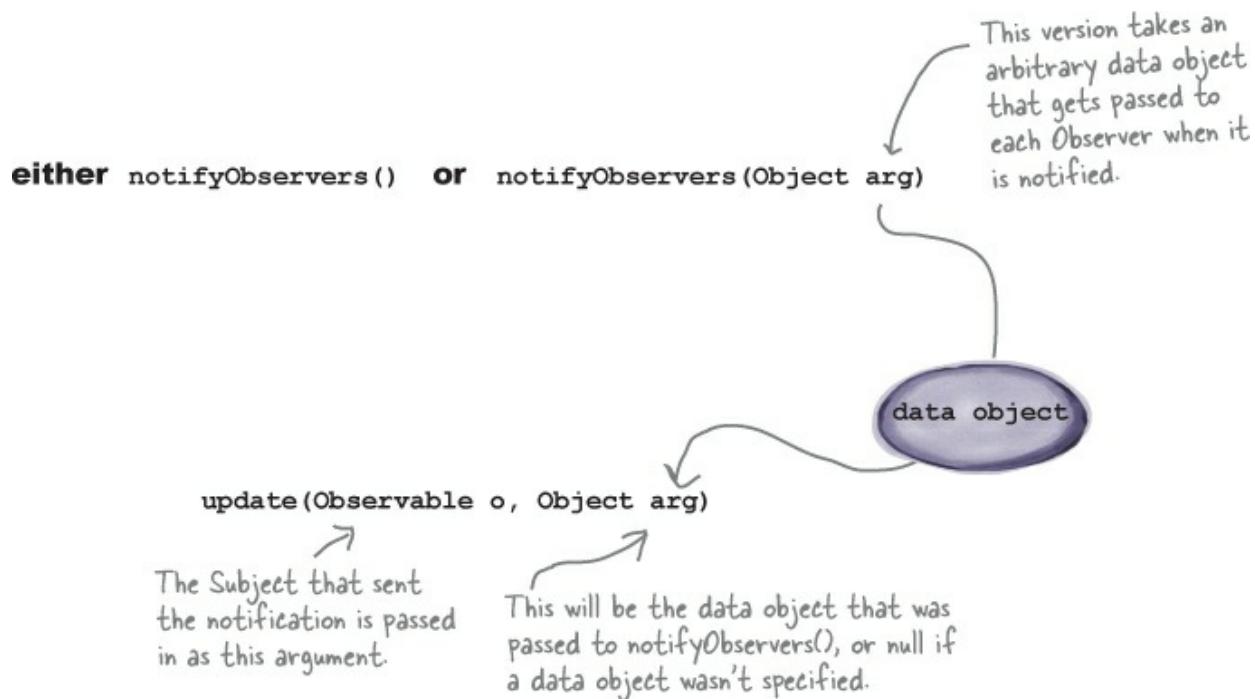
For an Object to become an observer...

As usual, implement the Observer interface (this time the `java.util.Observer` interface) and call `addObserver()` on any `Observable` object. Likewise, to remove yourself as an observer, just call `deleteObserver()`.

For the Observable to send notifications...

First of all you need to be `Observable` by extending the `java.util.Observable` superclass. From there it is a two-step process:

- ① You first must call the `setChanged()` method to signify that the state has changed in your object.
- ② Then, call one of two `notifyObservers()` methods:



For an Observer to receive notifications...

It implements the `update` method, as before, but the signature of the method is a bit different:

If you want to “push” data to the observers, you can pass the data as a data object to the `notifyObservers(arg)` method. If not, then the Observer has to

“pull” the data it wants from the Observable object passed to it. How? Let’s rework the Weather Station and you’ll see.



The `setChanged()` method is used to signify that the state has changed and that `notifyObservers()`, when it is called, should update its observers. If `notifyObservers()` is called without first calling `setChanged()`, the observers will NOT be notified. Let’s take a look behind the scenes of Observable to see how this works:

BEHIND THE SCENES

Pseudocode for the Observable class.

```

setChanged() {
    changed = true
}

notifyObservers(Object arg) {
    if (changed) {
        for every observer on the list {
            call update (this, arg)
        }
    }
    changed = false
}

notifyObservers() {
    notifyObservers(null)
}

```

The `setChanged()` method sets a changed flag to true.

`notifyObservers()` only notifies its observers if the changed flag is TRUE.

And after it notifies the observers, it sets the changed flag back to false.

Why is this necessary? The `setChanged()` method is meant to give you more flexibility in how you update observers by allowing you to optimize the notifications. For example, in our Weather Station, imagine if our measurements were so sensitive that the temperature readings were constantly fluctuating by a few tenths of a degree. That might cause the `WeatherData` object to send out notifications constantly. Instead, we might want to send out notifications only if the temperature changes more than half a degree and we could call `setChanged()` only after that happened.

You might not use this functionality very often, but it's there if you need it. In either case, you need to call `setChanged()` for notifications to work. If this functionality is something that is useful to you, you may also want to use the `clearChanged()` method, which sets the changed state back to false, and the `hasChanged()` method, which tells you the current state of the changed flag.

Reworking the Weather Station with the built-in support

First, let's rework `WeatherData` to use `java.util.Observable`

- 1 Make sure we are importing the right Observable.

```
import java.util.Observable;
```

- 2 We are now subclassing Observable.

```
public class WeatherData extends Observable {  
    private float temperature;  
    private float humidity;  
    private float pressure;  
  
    public WeatherData() {}  
  
    public void measurementsChanged() {  
        setChanged();  
        notifyObservers(); *  
    }  
  
    public void setMeasurements(float temperature, float humidity, float pressure) {  
        this.temperature = temperature;  
        this.humidity = humidity;  
        this.pressure = pressure;  
        measurementsChanged();  
    }  
  
    public float getTemperature() {  
        return temperature;  
    }  
  
    public float getHumidity() {  
        return humidity;  
    }  
  
    public float getPressure() {  
        return pressure;  
    }  
}
```

- 3 We don't need to keep track of our observers anymore, or manage their registration and removal (the superclass will handle that), so we've removed the registerObserver(), removeObserver() and notifyObservers() methods.

- 4 Our constructor no longer needs to create a data structure to hold Observers.

- * Notice we aren't sending a data object with the notifyObservers() call. That means we're using the PULL model.

- 5 We now first call setChanged() to indicate the state has changed before calling notifyObservers().

- 6 These methods aren't new, but because we are going to use "pull" we thought we'd remind you they are here. The Observers will use them to get at the WeatherData object's state.

Now, let's rework the CurrentConditionsDisplay

1 Again, make sure we are importing the right Observer/Observable.

```
import java.util.Observable;
import java.util.Observer;
```

2 We now are implementing the Observer interface from java.util.

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    Observable observable;
    private float temperature;
    private float humidity;

    public CurrentConditionsDisplay(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }

}
```

3 Our constructor now takes an Observable and we use this to add the current conditions object as an Observer.

```
public void update(Observable obs, Object arg) {
    if (obs instanceof WeatherData) {
        WeatherData weatherData = (WeatherData) obs;
        this.temperature = weatherData.getTemperature();
        this.humidity = weatherData.getHumidity();
        display();
    }
}

public void display() {
    System.out.println("Current conditions: " + temperature
        + "F degrees and " + humidity + "% humidity");
}
```

4 We've changed the update() method to take both an Observable and the optional data argument.

5 In update(), we first make sure the observable is of type WeatherData and then we use its getter methods to obtain the temperature and humidity measurements. After that we call display().

CODE MAGNETS

The ForecastDisplay class is all scrambled up on the fridge. Can you reconstruct the code snippets to make it work? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need!

```
public ForecastDisplay(Observable  
observable) {
```

```
    display();
```

```
    observable.addObserver(this);
```

```
if (observable instanceof WeatherData) {
```

```
public class ForecastDisplay implements  
Observer, DisplayElement {
```

```
    public void display() {  
        // display code here  
    }
```

```
    lastPressure = currentPressure;  
    currentPressure = weatherData.getPressure();
```

```
    private float currentPressure = 29.92f;  
    private float lastPressure;
```

```
    WeatherData weatherData = (WeatherData)observable;
```

```
    public void update(Observable observable,  
Object arg) {
```

```
        import java.util.Observable;  
        import java.util.Observer;
```

Running the new code

Just to be sure, let's run the new code...

```
File Edit Window Help TryThisAtHome
%java WeatherStation
Forecast: Improving weather on the way!
Avg/Max/Min temperature = 80.0/80.0/80.0
Current conditions: 80.0F degrees and 65.0% humidity
Forecast: Watch out for cooler, rainy weather
Avg/Max/Min temperature = 81.0/82.0/80.0
Current conditions: 82.0F degrees and 70.0% humidity
Forecast: More of the same
Avg/Max/Min temperature = 80.0/82.0/78.0
Current conditions: 78.0F degrees and 90.0% humidity
%
```

Hmm, do you notice anything different? Look again...

You'll see all the same calculations, but mysteriously, the order of the text output is different. Why might this happen? Think for a minute before reading on...

Never depend on order of evaluation of the Observer notifications

The `java.util.Observable` has implemented its `notifyObservers()` method such that the Observers are notified in a *different* order than our own implementation. Who's right? Neither; we just chose to implement things in different ways.

What would be incorrect, however, is if we wrote our code to *depend* on a specific notification order. Why? Because if you need to change Observable/Observer implementations, the order of notification could change and your application would produce incorrect results. Now that's definitely not what we'd consider loosely coupled.



The dark side of `java.util.Observable`

Yes, good catch. As you've noticed, `Observable` is a class, not an *interface*, and worse, it doesn't even *implement* an interface. Unfortunately, the `java.util.Observable` implementation has a number of problems that limit its usefulness and reuse. That's not to say it doesn't provide some utility, but there are some large potholes to watch out for.

Observable is a class

You already know from our principles this is a bad idea, but what harm does it really cause?

First, because `Observable` is a *class*, you have to *subclass* it. That means you can't add on the `Observable` behavior to an existing class that already extends another superclass. This limits its reuse potential (and isn't that why we are

using patterns in the first place?).

Second, because there isn't an Observable interface, you can't even create your own implementation that plays well with Java's built-in Observer API. Nor do you have the option of swapping out the java.util implementation for another (say, a new, multithreaded implementation).

Observable protects crucial methods

If you look at the Observable API, the `setChanged()` method is protected. So what? Well, this means you can't call `setChanged()` unless you've subclassed Observable. This means you can't even create an instance of the Observable class and compose it with your own objects, you *have* to subclass. The design violates a second design principle here...*favor composition over inheritance*.

What to do?

Observable *may* serve your needs if you can extend `java.util.Observable`. On the other hand, you may need to roll your own implementation as we did at the beginning of the chapter. In either case, you know the Observer Pattern well and you're in a good position to work with any API that makes use of the pattern.

Other places you'll find the Observer Pattern in the JDK

The `java.util` implementation of Observer/Observable is not the only place you'll find the Observer Pattern in the JDK; both JavaBeans and Swing also provide their own implementations of the pattern. At this point you understand enough about Observer to explore these APIs on your own; however, let's do a quick, simple Swing example just for the fun of it.

NOTE

If you're curious about the Observer Pattern in JavaBeans, check out the `PropertyChangeListener` interface.

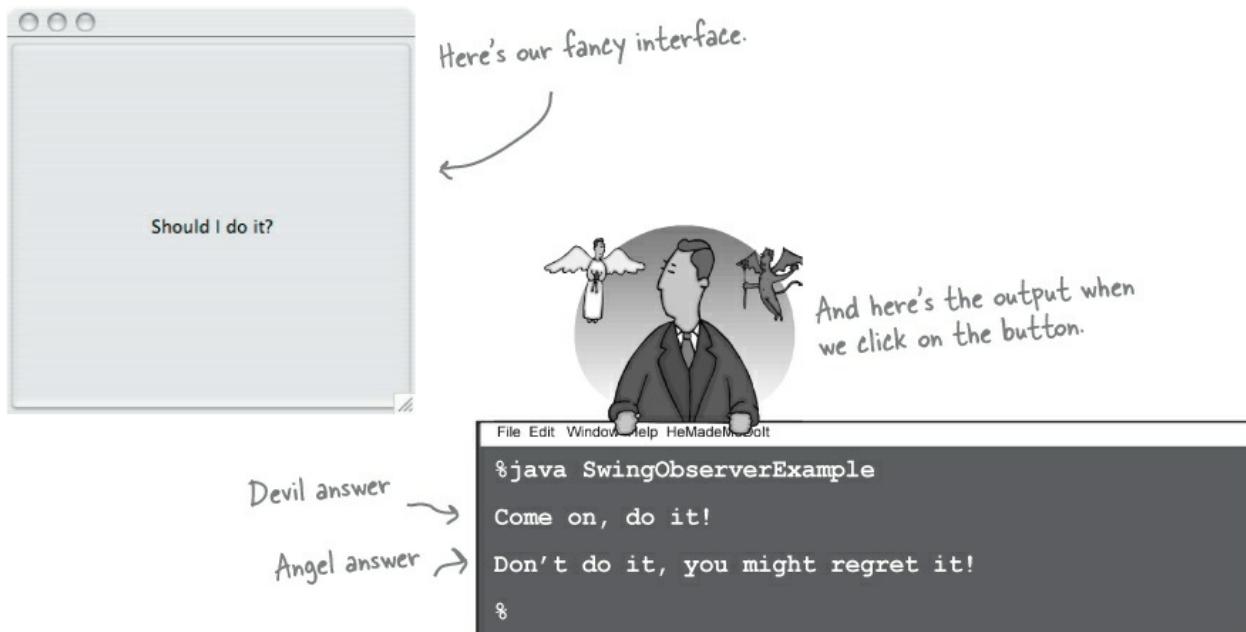
A little background...

Let's take a look at a simple part of the Swing API, the JButton. If you look under the hood at JButton's superclass, AbstractButton, you'll see that it has a lot of add/ remove listener methods. These methods allow you to add and

remove observers, or, as they are called in Swing, listeners, to listen for various types of events that occur on the Swing component. For instance, an ActionListener lets you “listen in” on any types of actions that might occur on a button, like a button press. You’ll find various types of listeners all over the Swing API.

A little life-changing application

Okay, our application is pretty simple. You’ve got a button that says “Should I do it?” and when you click on that button the listeners (observers) get to answer the question in any way they want. We’re implementing two such listeners, called the AngelListener and the DevilListener. Here’s how the application behaves:



And the code...

This life-changing application requires very little code. All we need to do is create a JButton object, add it to a JFrame and set up our listeners. We’re going to use inner classes for the listeners, which is a common technique in Swing programming. If you aren’t up on inner classes or Swing, you might want to review the “Getting GUI” chapter of *Head First Java*.

```

public class SwingObserverExample {
    JFrame frame;
    public static void main(String[] args) {
        SwingObserverExample example = new SwingObserverExample();
        example.go();
    }
    public void go() {
        frame = new JFrame();

        JButton button = new JButton("Should I do it?");
        button.addActionListener(new AngelListener());
        button.addActionListener(new DevilListener());

        // Set frame properties here
    }

    class AngelListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Don't do it, you might regret it!");
        }
    }

    class DevilListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Come on, do it!");
        }
    }
}

```

Simple Swing application that just creates a frame and throws a button in it.

Makes the devil and angel objects listeners (observers) of the button.

Here are the class definitions for the observers, defined as inner classes (but they don't have to be).

Rather than update(), the actionPerformed() method gets called when the state in the subject (in this case the button) changes.



I've been using lambda
expressions in place of simple
action listeners in my Swing
code. Am I still using the
Observer Pattern?

o
o

NOTE

Lambda expressions were added in Java 8. If you aren't familiar with them, don't worry about it; you can continue using inner classes for your Swing observers.

Yes, you're still using the Observer Pattern. By using a lambda expression rather than an inner class, you're just skipping the step of creating an ActionListener object. With a lambda expression, you create a function object instead, and this function object is the observer. When you pass that function object to addActionListener(), Java ensures its signature matches actionPerformed(), the one method in the ActionListener interface.

Later, when the button is clicked, the button object notifies its observers — including the function objects created by the lambda expressions — that it's been clicked, and calls each listener's actionPerformed() method.

Let's take a look at how you'd use lambda expressions as observers to simplify our previous code:

The updated code, using lambda expressions

```
public class SwingObserverExample {  
    JFrame frame;  
    public static void main(String[] args) {  
        SwingObserverExample example = new SwingObserverExample();  
        example.go();  
    }  
    public void go() {  
        frame = new JFrame();  
  
        JButton button = new JButton("Should I do it?");  
        button.addActionListener(event ->  
            System.out.println("Don't do it, you might regret it!"));  
        button.addActionListener(event ->  
            System.out.println("Come on, do it!"));  
  
        // Set frame properties here  
    }  
}
```

We've removed the two ActionListener classes (DevilListener and AngelListener) completely.

We've replaced the AngelListener and DevilListener objects with lambda expressions that implement the same functionality that we had before.

When you click the button, the function objects created by the lambda expressions are notified and the method they implement is run.

Using lambda expressions makes this code a lot more concise.

For more on lambda expressions, check out the Java docs, and Chapter 6.

Tools for your Design Toolbox

Welcome to the end of **Chapter 2**. You've added a few new things to your OO toolbox...

OO Basics

Abstraction

Iteration

Mutation

Encapsulation

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Here's your newest principle. Remember, loosely coupled designs are much more flexible and resilient to change.

OO Patterns

Strategy
encapsulation
interface
vary

Observer - defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

A new pattern for communicating state to a set of objects in a loosely coupled manner. We haven't seen the last of the Observer Pattern—just wait until we talk about MVC!

BULLET POINTS

- The Observer Pattern defines a one-to-many relationship between objects.
- Subjects, or as we also know them, Observables, update Observers using a common interface.
- Observers are loosely coupled in that the Observable knows nothing about them, other than that they implement the Observer interface.
- You can push or pull data from the Observable when using the pattern (pull is considered more “correct”).
- Don’t depend on a specific order of notification for your Observers.
- Java has several implementations of the Observer Pattern, including the general purpose `java.util.Observable`.
- Watch out for issues with the `java.util.Observable` implementation.
- Don’t be afraid to create your own Observable implementation if needed.
- Swing makes heavy use of the Observer Pattern, as do many GUI frameworks.
- You’ll also find the pattern in many other places, including JavaBeans and RMI.

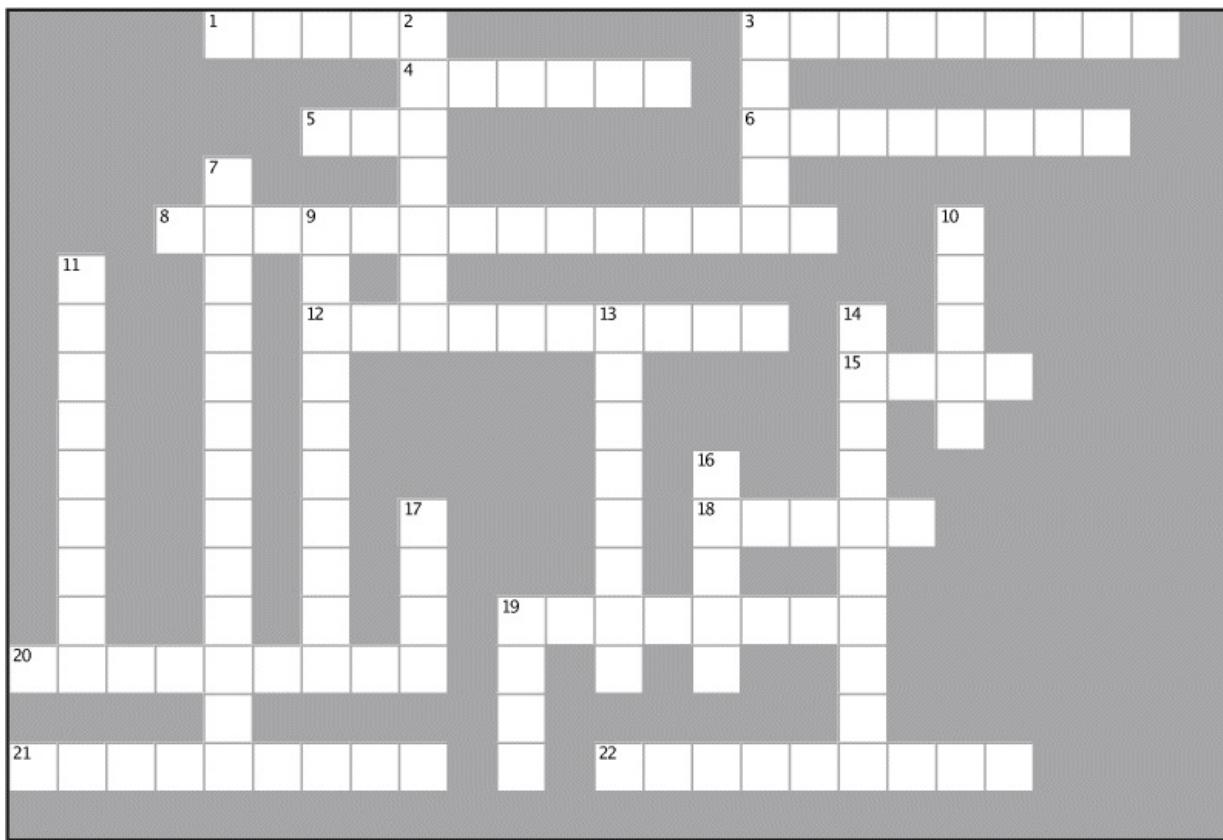
DESIGN PRINCIPLE CHALLENGE

For each design principle, describe how the Observer Pattern makes use of the principle.

DESIGN PRINCIPLE <i>Identify the aspects of your application that vary and separate them from what stays the same.</i>	<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>
DESIGN PRINCIPLE <i>Program to an interface, not an implementation.</i>	<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>
DESIGN PRINCIPLE <i>Favor composition over inheritance.</i>	<u>This is a hard one, hint: think about how observers and subjects work together.</u> <hr/> <hr/> <hr/> <hr/> <hr/>

DESIGN PATTERNS CROSSWORD

Time to give your right brain something to do again! This time all of the solution words are from [Chapter 2](#).



Across	Down
1. Observable is a _____, not an interface.	2. Ron was both an Observer and a _____.
3. Devil and Angel are _____ to the button.	3. You want to keep your coupling _____.
4. Implement this method to get notified.	7. He says you should go for it.
5. Jill got one of her own.	9. _____ can manage your observers for you.
6. CurrentConditionsDisplay implements this interface.	10. Java framework with lots of Observers.
8. How to get yourself off the Observer list.	11. Weather-O-Rama's CEO named after this kind of storm.
12. You forgot this if you're not getting notified when you think you should be.	13. Observers like to be _____ when something new happens.
15. One Subject likes to talk to _____ observers.	14. The WeatherData class _____ the
18. Don't count on this for notification.	
19. Temperature, humidity and _____.	

- | | |
|--|--|
| 20. Observers are _____ on the Subject. | Subject interface. |
| 21. Program to an _____ not an implementation. | 16. He didn't want any more ints, so he removed himself. |
| 22. A Subject is similar to a _____. | 17. CEO almost forgot the _____ index display
19. Subject initially wanted to _____ all the data to Observer. |

SHARPEN YOUR PENCIL SOLUTION

Based on our first implementation, which of the following apply? (Choose all that apply.)

- | | |
|-------------------------------------|---|
| <input checked="" type="checkbox"/> | A. We are coding to concrete implementations, not interfaces. |
| <input checked="" type="checkbox"/> | B. For every new display element we need to alter code. |
| <input checked="" type="checkbox"/> | C. We have no way to add display elements at run time. |
| <input type="checkbox"/> | D. The display elements don't implement a common interface. |
| <input checked="" type="checkbox"/> | E. We haven't encapsulated what changes. |
| <input type="checkbox"/> | F. We are violating encapsulation of the WeatherData class. |

DESIGN PRINCIPLE CHALLENGE SOLUTION

DESIGN PRINCIPLE

Identify the aspects of your application that vary and separate them from what stays the same.

The thing that varies in the Observer Pattern
is the state of the Subject and the number and
types of Observers. With this pattern, you can
vary the objects that are dependent on the state
of the Subject, without having to change that
Subject. That's called planning ahead!

DESIGN PRINCIPLE

Program to an interface, not an implementation.

Both the Subject and Observer use interfaces.
The Subject keeps track of objects
implementing
the Observer interface, while the
observers
register with, and get notified by, the Subject

interface. As we've seen, this keeps things nice
and
loosely coupled.

DESIGN PRINCIPLE

Favor composition over inheritance.

The Observer Pattern uses composition to
compose

any number of Observers with their
Subjects.

These relationships aren't set up by some kind
of

inheritance hierarchy. No, they are set up at

runtime by
composition!

CODE MAGNETS SOLUTION

The ForecastDisplay class is all scrambled up on the fridge. Can you reconstruct the code snippets to make it work? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need! Here's our solution.

```

import java.util.Observable;
import java.util.Observer;

public class ForecastDisplay implements
Observer, DisplayElement {

    private float currentPressure = 29.92f;
    private float lastPressure;

    public ForecastDisplay(Observable
observable) {

        WeatherData weatherData =
(WeatherData)observable;

        observable.addObserver(this);
    }

    public void update(Observable observable,
Object arg) {

        if (observable instanceof WeatherData) {

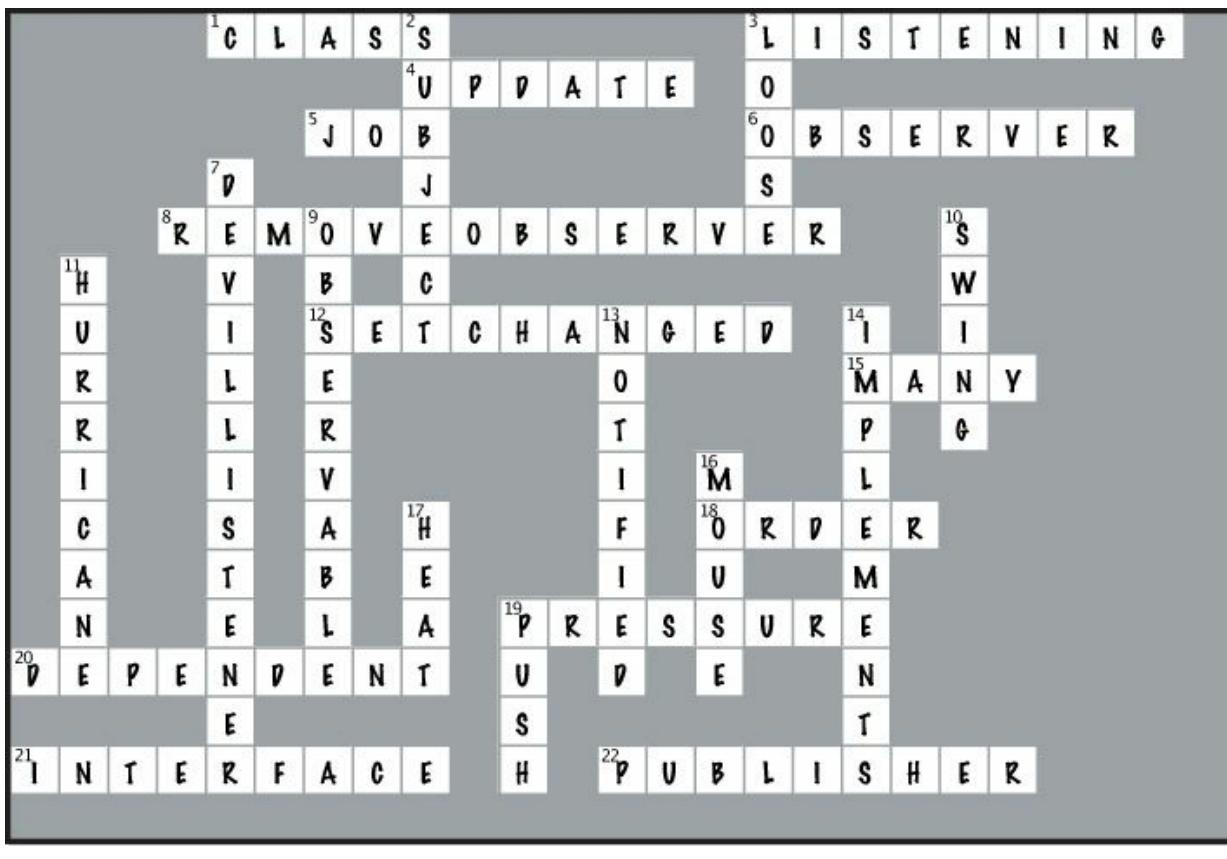
            lastPressure = currentPressure;
            currentPressure = weatherData.getPressure();

            display();
        }
    }

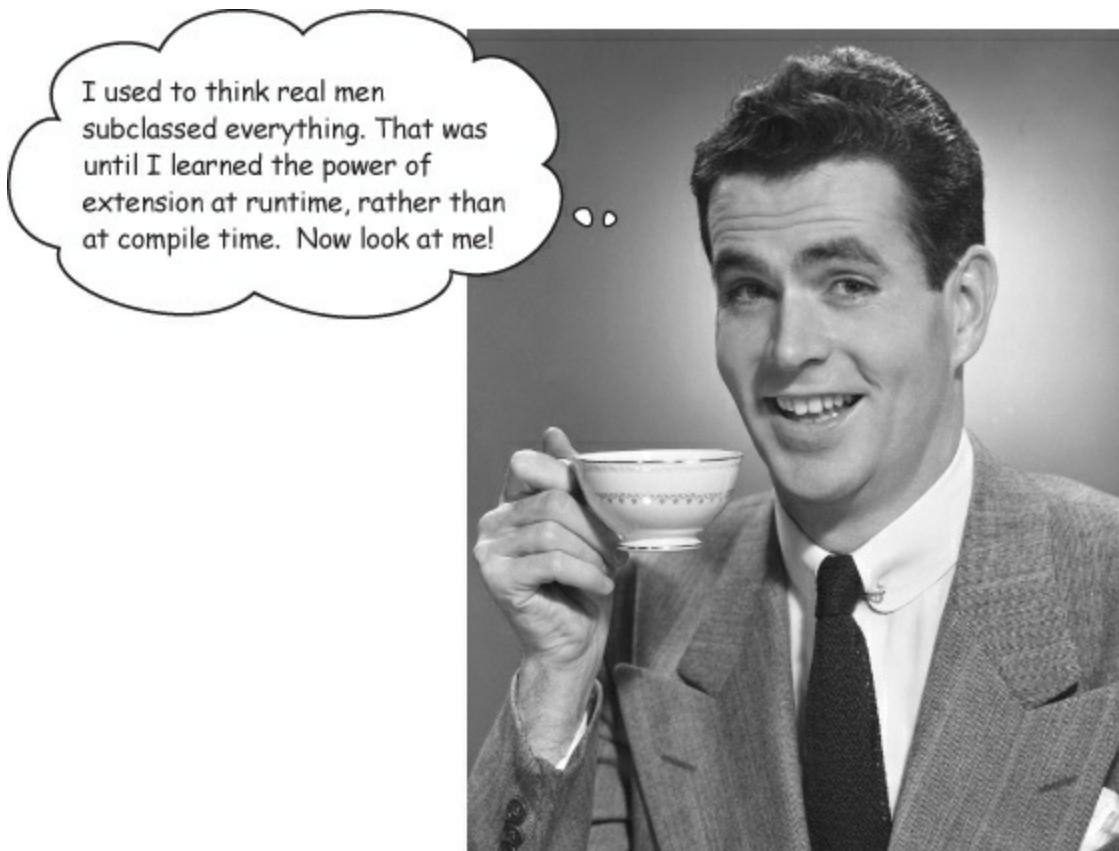
    public void display() {
        // display code here
    }
}

```

DESIGN PATTERNS CROSSWORD SOLUTION



Chapter 3. The Decorator Pattern: Decorating Objects



Just call this chapter “**Design Eye for the Inheritance Guy.**” We’ll re-examine the typical overuse of inheritance and you’ll learn how to decorate your classes at runtime using a form of object composition. Why? Once you know the techniques of decorating, you’ll be able to give your (or someone else’s) objects new responsibilities *without making any code changes to the underlying classes.*

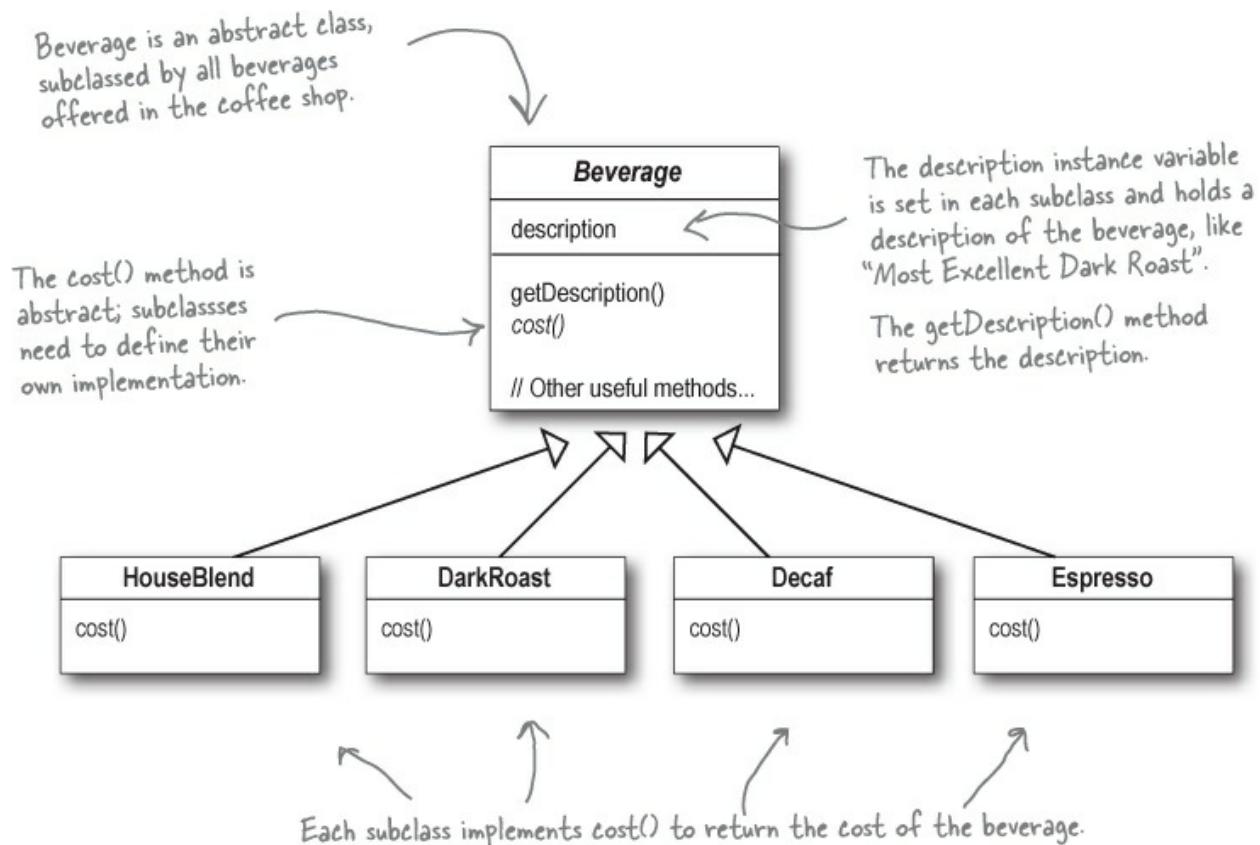
Welcome to Starbuzz Coffee



Starbuzz Coffee has made a name for itself as the fastest growing coffee shop around. If you've seen one on your local corner, look across the street; you'll see another one.

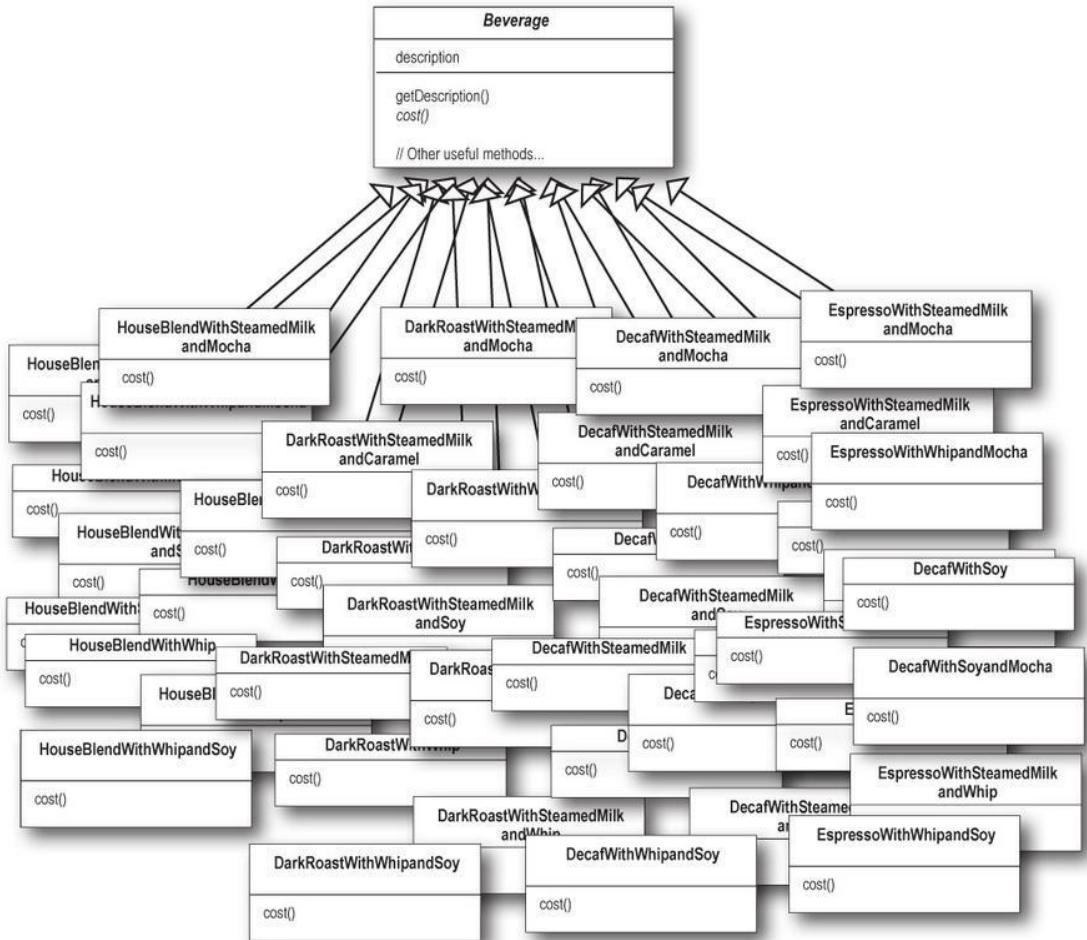
Because they've grown so quickly, they're scrambling to update their ordering systems to match their beverage offerings.

When they first went into business they designed their classes like this...



In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha (otherwise known as chocolate), and have it all topped off with whipped milk. Starbuzz charges a bit for each of these, so they really need to get them built into their order system.

Here's their first attempt...



Each cost method computes the cost of the coffee along with the other condiments in the order.



BRAIN POWER

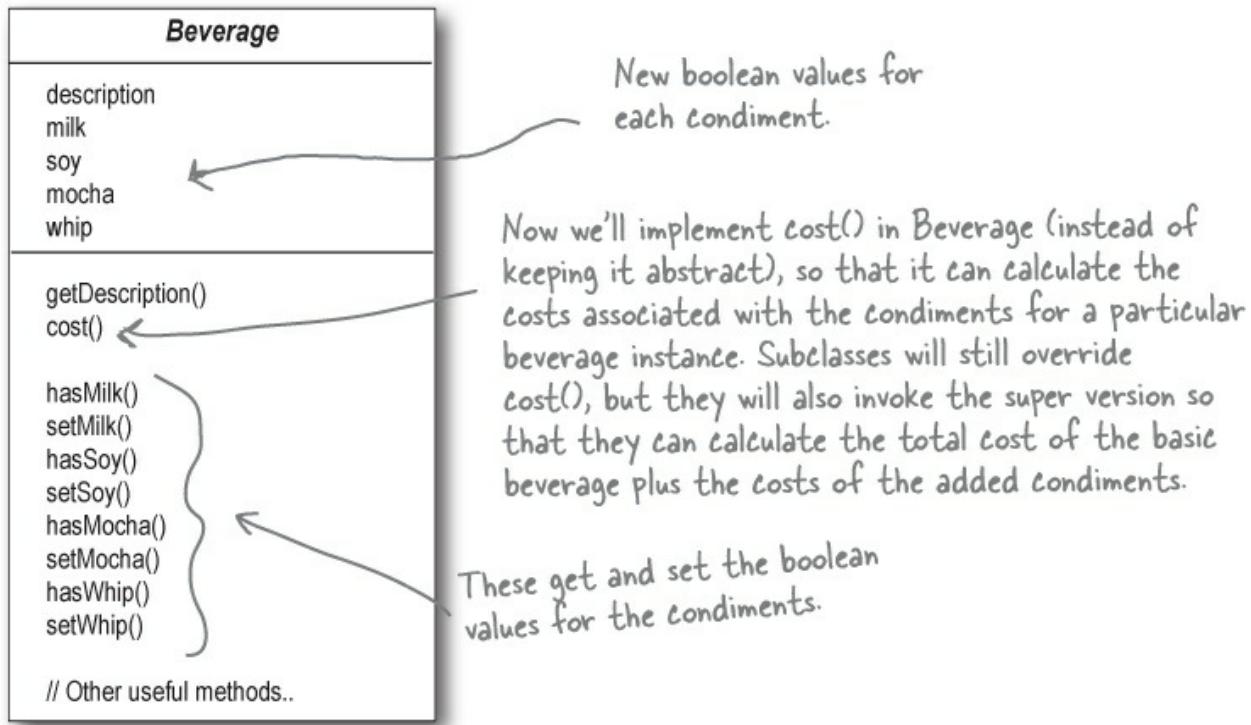
It's pretty obvious that Starbuzz has created a maintenance nightmare for themselves. What happens when the price of milk goes up? What do they do when they add a new caramel topping?

Thinking beyond the maintenance problem, which of the design principles that we've covered so far are they violating?

Hint: they're violating two of them in a big way!



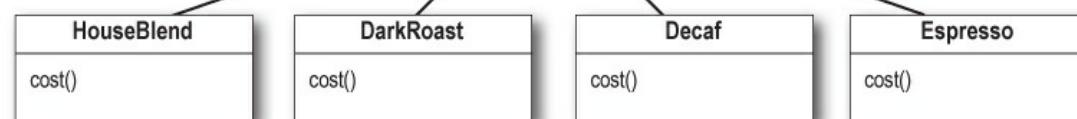
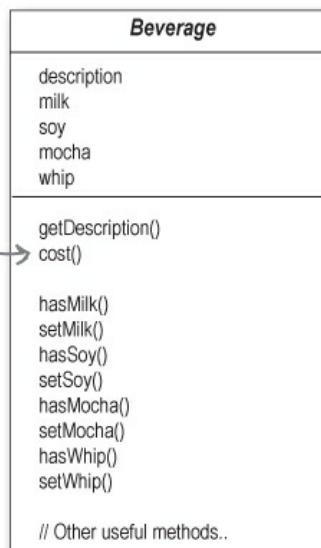
Well, let's give it a try. Let's start with the Beverage base class and add instance variables to represent whether or not each beverage has milk, soy, mocha, and whip...



Now let's add in the subclasses, one for each beverage on the menu:

The superclass cost() will calculate the costs for all of the condiments, while the overridden cost() in the subclasses will extend that functionality to include costs for that specific beverage type.

Each cost() method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of cost().



SHARPEN YOUR PENCIL

Write the cost() methods for the following classes (pseudo-Java is okay):

```
public class Beverage {  
    public double cost() {  
    }  
}
```

```
public class DarkRoast extends Beverage {  
    public DarkRoast() {  
        description = "Most Excellent Dark Roast";  
    }  
    public double cost() {  
    }  
}
```



See, five classes
total. This is definitely
the way to go.

I'm not so sure; I can see some
potential problems with this approach
by thinking about how the design might
need to change in the future.



SHARPEN YOUR PENCIL

What requirements or other factors might change that will impact this design?

Price changes for condiments will force us to alter existing code.

New condiments will force us to add new methods and alter the cost method in the superclass.

We may have new beverages. For some of these beverages (iced tea?), the condiments may not be appropriate, yet the Tea subclass will still inherit methods like hasWhip().

As we saw in Chapter 1, this is a very bad idea!

What if a customer wants a double mocha?

Your turn:

MASTER AND STUDENT...

Master: Grasshopper, it has been some time since our last meeting. Have you been deep in meditation on inheritance?

Student: Yes, Master. While inheritance is powerful, I have learned that it doesn't always lead to the most flexible or maintainable designs.

Master: Ah yes, you have made some progress. So, tell me, my student, how then will you achieve reuse if not through inheritance?

Student: Master, I have learned there are ways of "inheriting" behavior at runtime through composition and delegation.

Master: Please, go on...

Student: When I inherit behavior by subclassing, that behavior is set statically at compile time. In addition, all subclasses must inherit the same behavior. If however, I can extend an object's behavior through composition, then I can do this dynamically at runtime.

Master: Very good, Grasshopper, you are beginning to see the power of composition.

Student: Yes, it is possible for me to add multiple new responsibilities to objects through this technique, including responsibilities that were not even thought of by the designer of the superclass. And, I don't have to touch their code!

Master: What have you learned about the effect of composition on maintaining your code?

Student: Well, that is what I was getting at. By dynamically composing objects, I can add new functionality by writing new code rather than altering existing code. Because I'm not changing existing code, the chances of introducing bugs or causing unintended side effects in pre-existing code are much reduced.

Master: Very good. Enough for today, Grasshopper. I would like for you to go and meditate further on this topic... Remember, code should be closed (to change) like the lotus flower in the evening, yet open (to extension) like the lotus flower in the morning.

The Open-Closed Principle

Grasshopper is on to one of the most important design principles:

DESIGN PRINCIPLE

Classes should be open for extension, but closed for modification.



Come on in; we're *open*. Feel free to extend our classes with any new behavior you like. If your needs or requirements change (and we know they will), just go ahead and make your own extensions.



Sorry, we're *closed*. That's right, we spent a lot of time getting this code

correct and bug free, so we can't let you alter the existing code. It must remain closed to modification. If you don't like it, you can speak to the manager.

Our goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code. What do we get if we accomplish this? Designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements.

THERE ARE NO DUMB QUESTIONS

Q: Q: Open for extension and closed for modification? That sounds very contradictory. How can a design be both?

A: A: That's a very good question. It certainly sounds contradictory at first. After all, the less modifiable something is, the harder it is to extend, right?

As it turns out, though, there are some clever OO techniques for allowing systems to be extended, even if we can't change the underlying code. Think about the Observer Pattern (in [Chapter 2](#))... by adding new Observers, we can extend the Subject at any time, without adding code to the Subject. You'll see quite a few more ways of extending behavior with other OO design techniques.

Q: Q: Okay, I understand Observable, but how do I generally design something to be extensible, yet closed for modification?

A: A: Many of the patterns give us time-tested designs that protect your code from being modified by supplying a means of extension. In this chapter you'll see a good example of using the Decorator Pattern to follow the Open-Closed principle.

Q: Q: How can I make every part of my design follow the Open-Closed Principle?

A: A: Usually, you can't. Making OO design flexible and open to extension without the modification of existing code takes time and effort. In general, we don't have the luxury of tying down every part of our designs (and it would probably be wasteful). Following the Open-Closed Principle usually introduces new levels of abstraction, which adds complexity to our code. You want to concentrate on those areas that are most likely to change in your designs and apply the principles there.

Q: Q: How do I know which areas of change are more important?

A: A: That is partly a matter of experience in designing OO systems and also a matter of knowing the domain you are working in. Looking at other examples will help you learn to identify areas of change in your own designs.

While it may seem like a contradiction, there are techniques for allowing code to be extended without direct modification.

Be careful when choosing the areas of code that need to be extended; applying the Open-Closed Principle EVERYWHERE is wasteful and unnecessary, and can lead to complex, hard-to-understand code.

Meet the Decorator Pattern

Okay, we've seen that representing our beverage plus condiment pricing scheme with inheritance has not worked out very well — we get class explosions and rigid designs, or we add functionality to the base class that

isn't appropriate for some of the subclasses.

So, here's what we'll do instead: we'll start with a beverage and "decorate" it with the condiments at runtime. For example, if the customer wants a Dark Roast with Mocha and Whip, then we'll:

- ① Take a DarkRoast object**
- ② Decorate it with a Mocha object**
- ③ Decorate it with a Whip object**
- ④ Call the cost() method and rely on delegation to add on the condiment costs**

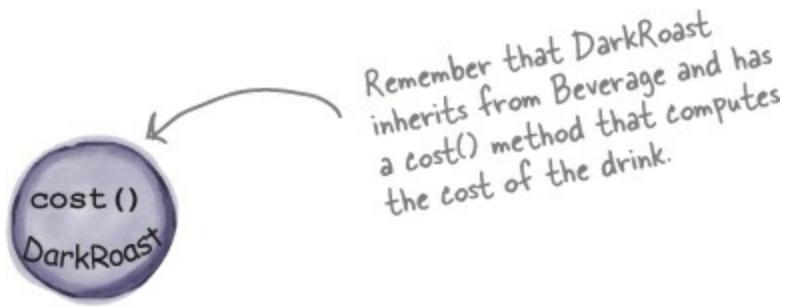
Okay, but how do you "decorate" an object, and how does delegation come into this? A hint: think of decorator objects as "wrappers." Let's see how this works...



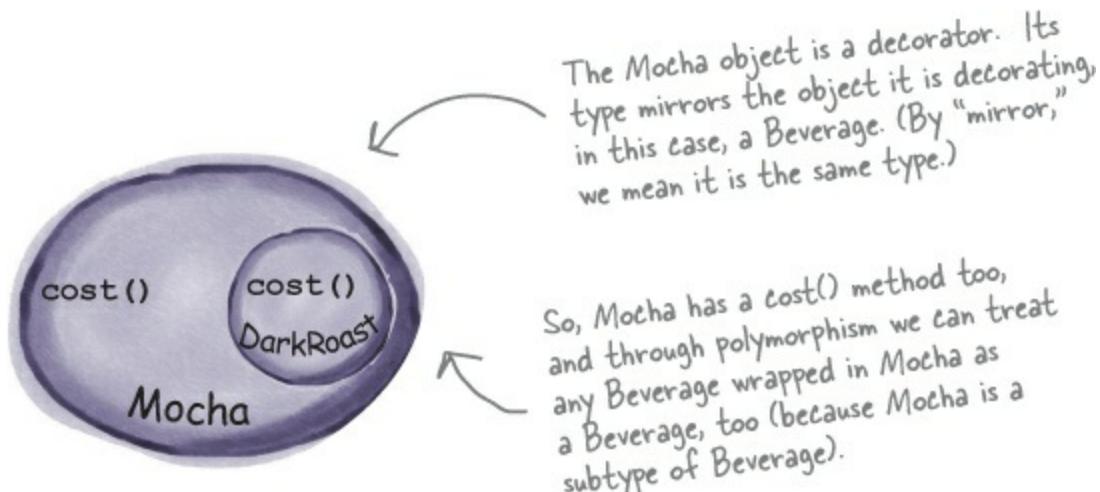
Okay, enough of the "Object Oriented Design Club." We have real problems here! Remember us? Starbuzz Coffee? Do you think you could use some of those design principles to actually help us?

Constructing a drink order with Decorators

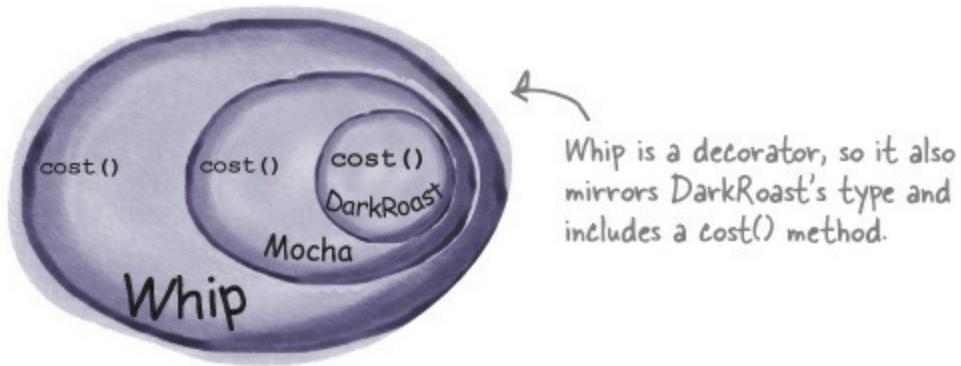
- ① We start with our DarkRoast object.



- ② The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.**



- ③ The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.**

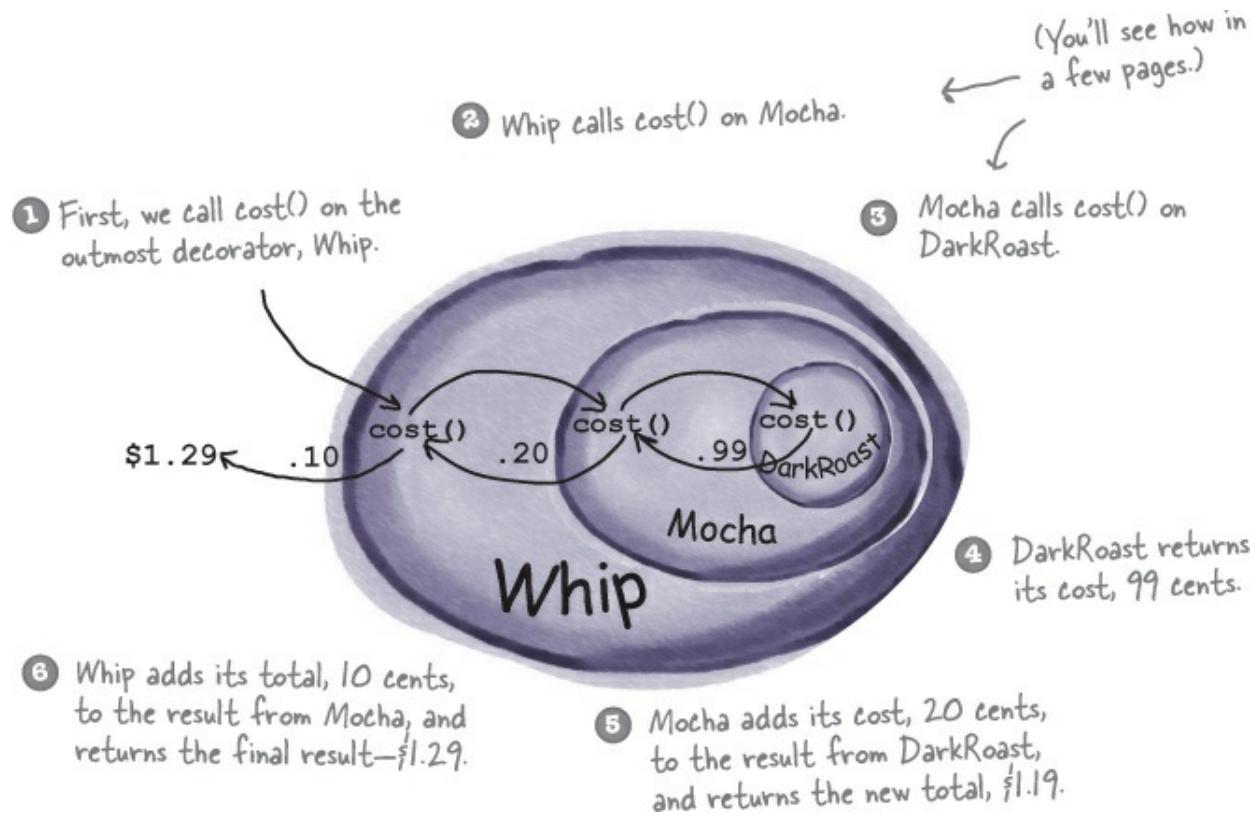


NOTE

So, a `DarkRoast` wrapped in `Mocha` and `Whip` is still a `Beverage` and we can do anything with it we can do with a `DarkRoast`, including call its `cost()` method.

- ④ Now it's time to compute the cost for the customer. We do this by**

calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.



Okay, here's what we know so far...

- Decorators have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.

NOTE

Key point!

- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.

Now let's see how this all really works by looking at the **Decorator Pattern** definition and writing some code.

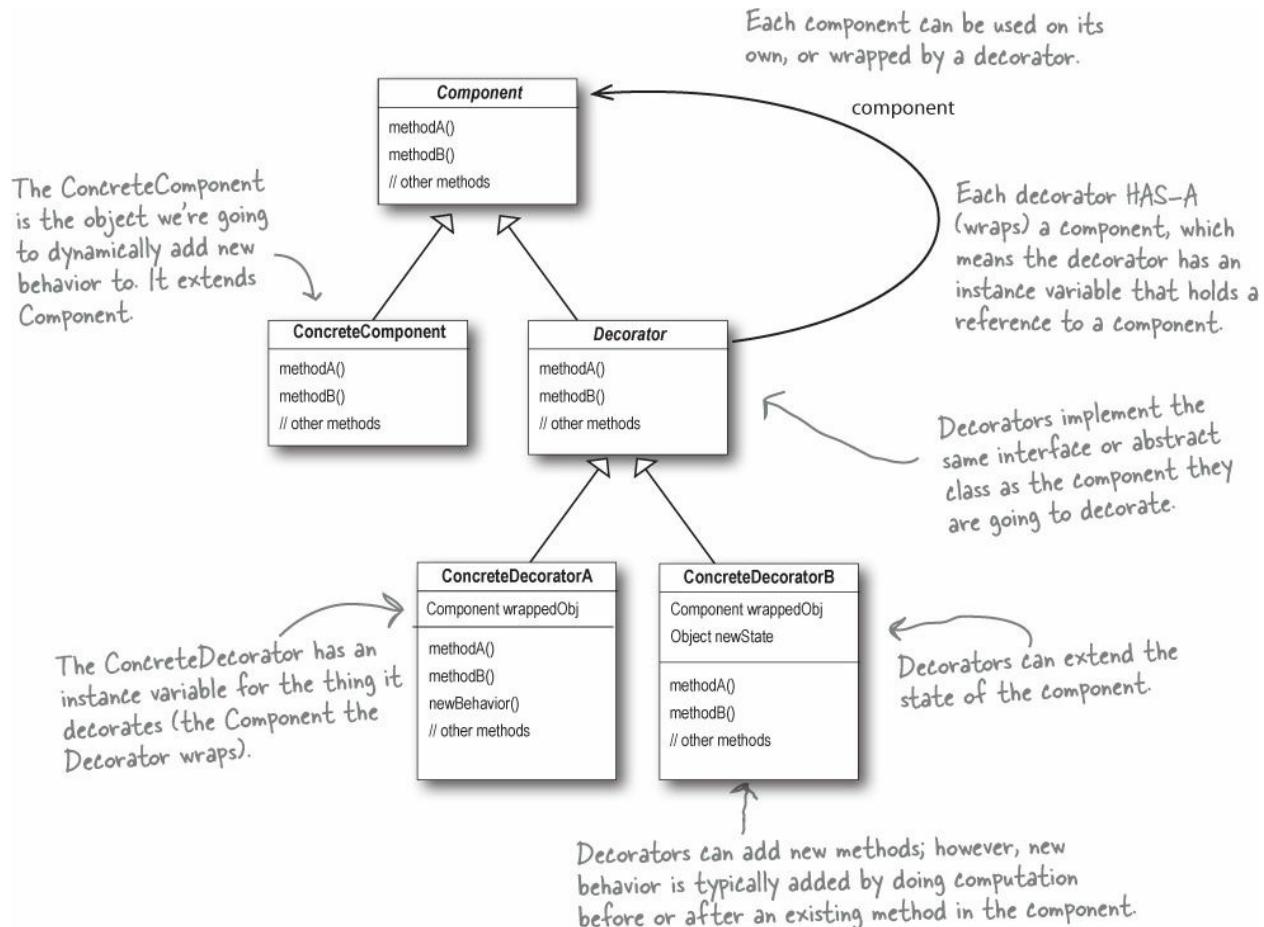
The Decorator Pattern defined

Let's first take a look at the Decorator Pattern description:

NOTE

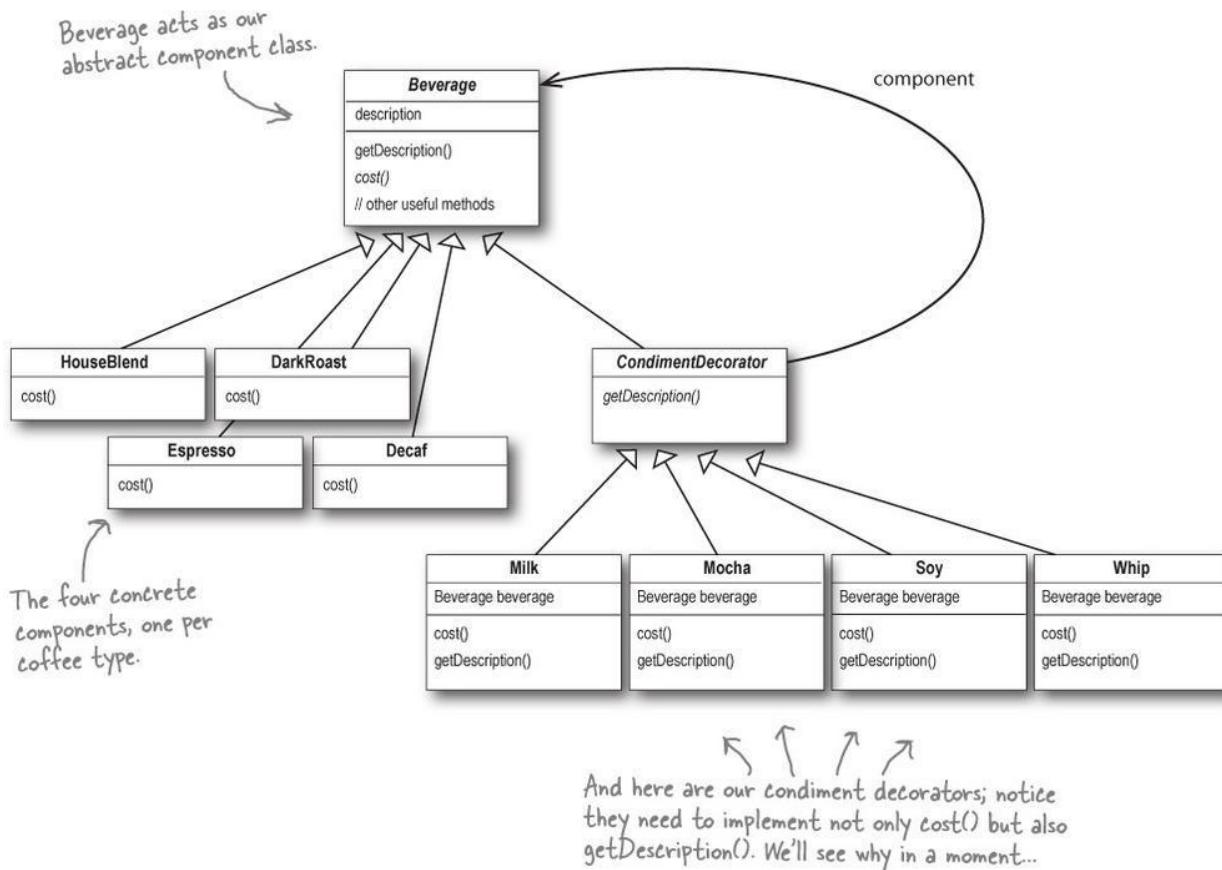
The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

While that describes the *role* of the Decorator Pattern, it doesn't give us a lot of insight into how we'd *apply* the pattern to our own implementation. Let's take a look at the class diagram, which is a little more revealing (on the next page we'll look at the same structure applied to the beverage problem).



Decorating our Beverages

Okay, let's work our Starbuzz beverages into this framework...



BRAIN POWER

Before going further, think about how you'd implement the `cost()` method of the coffees and the condiments. Also think about how you'd implement the `getDescription()` method of the condiments.

Cubicle Conversation

Some confusion over Inheritance versus Composition



Sue: What do you mean?

Mary: Look at the class diagram. The CondimentDecorator is extending the Beverage class. That's inheritance, right?

Sue: True. I think the point is that it's vital that the decorators have the same type as the objects they are going to decorate. So here we're using inheritance to achieve the *type matching*, but we aren't using inheritance to get *behavior*.

Mary: Okay, I can see how decorators need the same "interface" as the components they wrap because they need to stand in place of the component. But where does the behavior come in?

Sue: When we compose a decorator with a component, we are adding new behavior. We are acquiring new behavior not by inheriting it from a superclass, but by composing objects together.

Mary: Okay, so we're subclassing the abstract class Beverage in order to have the correct type, not to inherit its behavior. The behavior comes in through the composition of decorators with the base components as well as

other decorators.

Sue: That's right.

Mary: Ooooh, I see. And because we are using object composition, we get a whole lot more flexibility about how to mix and match condiments and beverages. Very smooth.

Sue: Yes, if we rely on inheritance, then our behavior can only be determined statically at compile time. In other words, we get only whatever behavior the superclass gives us or that we override. With composition, we can mix and match decorators any way we like... *at runtime*.

Mary: And as I understand it, we can implement new decorators at any time to add new behavior. If we relied on inheritance, we'd have to go in and change existing code any time we wanted new behavior.

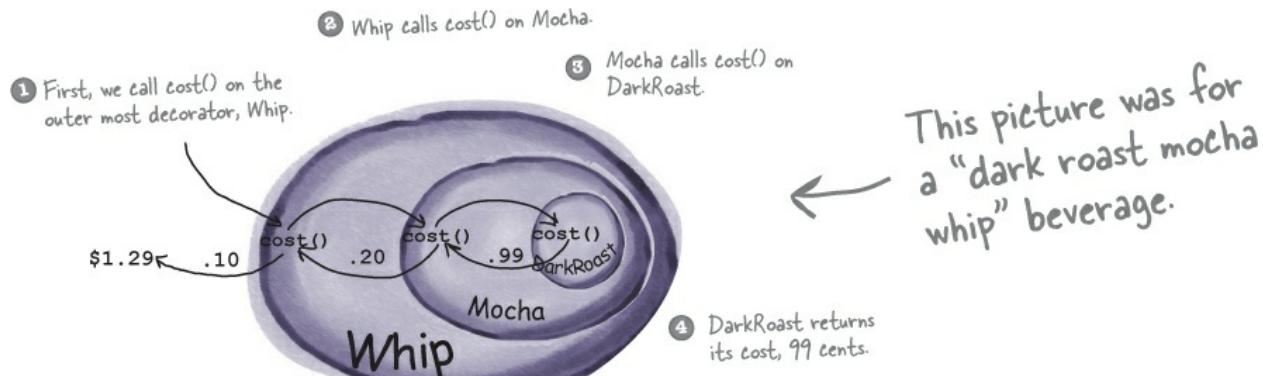
Sue: Exactly.

Mary: I just have one more question. If all we need to inherit is the type of the component, how come we didn't use an interface instead of an abstract class for the Beverage class?

Sue: Well, remember, when we got this code, Starbuzz already *had* an abstract Beverage class. Traditionally the Decorator Pattern does specify an abstract component, but in Java, obviously, we could use an interface. But we always try to avoid altering existing code, so don't "fix" it if the abstract class will work just fine.

New barista training

Make a picture for what happens when the order is for a "double mocha soy latte with whip" beverage. Use the menu to get the correct prices, and draw your picture using the same format we used earlier (from a few pages back):



This picture was for a "dark roast mocha whip" beverage.

- 5 `Mocha` adds its cost, 20 cents, to the result from `DarkRoast`, and returns the new total, \$1.19.

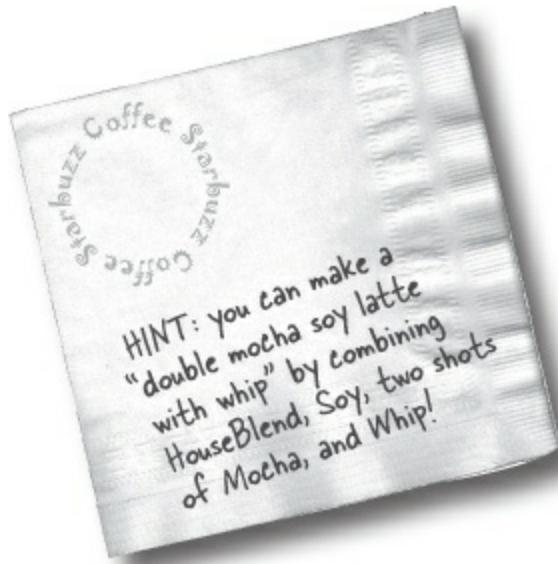
Okay, I need for you to make me a double mocha, soy latte with whip.



Starbuzz Coffee	
Coffees	
House Blend	.89
Dark Roast	.99
Decaf	1.05
Espresso	1.99
Condiments	
Steamed Milk	.10
Mocha	.20
Soy	.15
Whip	.10

SHARPEN YOUR PENCIL

Draw your picture here.



Writing the Starbuzz code

It's time to whip this design into some real code.

Let's start with the Beverage class, which doesn't need to change from Starbuzz's original design. Let's take a look:



```

public abstract class Beverage {
    String description = "Unknown Beverage";

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}

```

Beverage is an abstract class with the two methods `getDescription()` and `cost()`.

`getDescription` is already implemented for us, but we need to implement `cost()` in the subclasses.

Beverage is simple enough. Let's implement the abstract class for the Condiments (Decorator) as well:

```

public abstract class CondimentDecorator extends Beverage {
    public abstract String getDescription();
}

```

First, we need to be interchangeable with a Beverage, so we extend the Beverage class.

We're also going to require that the condiment decorators all reimplement the `getDescription()` method. Again, we'll see why in a sec...

Coding beverages

Now that we've got our base classes out of the way, let's implement some beverages. We'll start with Espresso. Remember, we need to set a description for the specific beverage and also implement the `cost()` method.

```
public class Espresso extends Beverage {
```

```
    public Espresso() {
        description = "Espresso";
    }

    public double cost() {
        return 1.99;
    }
}
```

First we extend the Beverage class, since this is a beverage.

To take care of the description, we set this in the constructor for the class. Remember, the description instance variable is inherited from Beverage.

Finally, we need to compute the cost of an Espresso. We don't need to worry about adding in condiments in this class, we just need to return the price of an Espresso: \$1.99.

```
public class HouseBlend extends Beverage {
```

```
    public HouseBlend() {
        description = "House Blend Coffee";
    }

    public double cost() {
        return .89;
    }
}
```

Okay, here's another Beverage. All we do is set the appropriate description, "House Blend Coffee," and then return the correct cost: 89¢.

You can create the other two Beverage classes (DarkRoast and Decaf) in exactly the same way.

Starbuzz Coffee		
Coffees		
House Blend		.89
Dark Roast		.99
Decaf		1.05
Espresso		1.99
Condiments		
Steamed Milk		.10
Mocha		.20
Soy		.15
Whip		.10

Coding condiments

If you look back at the Decorator Pattern class diagram, you'll see we've now written our abstract component (Beverage), we have our concrete components (HouseBlend), and we have our abstract decorator (CondimentDecorator). Now it's time to implement the concrete

decorators. Here's Mocha:

```
Mocha is a decorator, so we
extend CondimentDecorator.  

↓  

public class Mocha extends CondimentDecorator {  

    Beverage beverage;  

    public Mocha(Beverage beverage) {  

        this.beverage = beverage;  

    }  

    public String getDescription() {  

        return beverage.getDescription() + ", Mocha";  

    }  

    public double cost() {  

        return beverage.cost() + .20;  

    }  

}  

Now we need to compute the cost of our beverage
with Mocha. First, we delegate the call to the
object we're decorating, so that it can compute the
cost; then, we add the cost of Mocha to the result.
```

Remember, CondimentDecorator
extends Beverage.

We're going to instantiate Mocha with a
reference to a Beverage using:
(1) An instance variable to hold the
beverage we are wrapping.
(2) A way to set this instance
variable to the object we are wrapping.
Here, we're going to pass the beverage
we're wrapping to the decorator's
constructor.

We want our description to not only
include the beverage – say "Dark
Roast" – but also to include each
item decorating the beverage (for
instance, "Dark Roast, Mocha"). So
we first delegate to the object we are
decorating to get its description, then
append ", Mocha" to that description.

On the next page we'll actually instantiate the beverage and wrap it with all its condiments (decorators), but first...

EXERCISE

Write and compile the code for the other Soy and Whip condiments. You'll need them to finish and test the application.

Serving some coffees

Congratulations. It's time to sit back, order a few coffees, and marvel at the flexible design you created with the Decorator Pattern.

Here's some test code*to make orders:

```

public class StarbuzzCoffee {
    public static void main(String args[]) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription()
            + " $" + beverage.cost());
        Beverage beverage2 = new DarkRoast(); ← Order up an espresso, no condiments,
        beverage2 = new Mocha(beverage2); ← and print its description and cost.
        beverage2 = new Mocha(beverage2); ← Make a DarkRoast object
        beverage2 = new Whip(beverage2); ← Wrap it with a Mocha.
        System.out.println(beverage2.getDescription()
            + " $" + beverage2.cost());
        Beverage beverage3 = new HouseBlend(); ← Wrap it in a second Mocha.
        beverage3 = new Soy(beverage3); ← beverage3 = new Mocha(beverage3);
        beverage3 = new Whip(beverage3); ← Wrap it in a Whip.
        System.out.println(beverage3.getDescription()
            + " $" + beverage3.cost());
    }
}

```

* We're going to see a much better way of creating decorated objects when we cover the Factory and Builder Design Patterns. Please note that the Builder Pattern is covered in the Appendix.

Now, let's get those orders in:

```

File Edit Window Help CloudsInMyCoffee
% java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
%

```

THERE ARE NO DUMB QUESTIONS

Q: Q: I'm a little worried about code that might test for a specific concrete component — say, HouseBlend — and do something, like issue a discount. Once I've wrapped the HouseBlend with decorators, this isn't going to work anymore.

A: A: That is exactly right. If you have code that relies on the concrete component's type, decorators will break that

code. As long as you only write code against the abstract component type, the use of decorators will remain transparent to your code. However, once you start writing code against concrete components, you'll want to rethink your application design and your use of decorators.

Q: Q: Wouldn't it be easy for some client of a beverage to end up with a decorator that isn't the outermost decorator? Like if I had a DarkRoast with Mocha, Soy, and Whip, it would be easy to write code that somehow ended up with a reference to Soy instead of Whip, which means it would not include Whip in the order.

A: A: You could certainly argue that you have to manage more objects with the Decorator Pattern and so there is an increased chance that coding errors will introduce the kinds of problems you suggest. However, decorators are typically created by using other patterns like Factory and Builder. Once we've covered these patterns, you'll see that the creation of the concrete component with its decorator is "well encapsulated" and doesn't lead to these kinds of problems.

Q: Q: Can decorators know about the other decorations in the chain? Say I wanted my `getDescription()` method to print "Whip, Double Mocha" instead of "Mocha, Whip, Mocha." That would require that my outermost decorator know all the decorators it is wrapping.

A: A: Decorators are meant to add behavior to the object they wrap. When you need to peek at multiple layers into the decorator chain, you are starting to push the decorator beyond its true intent. Nevertheless, such things are possible. Imagine a CondimentPrettyPrint decorator that parses the final description and can print "Mocha, Whip, Mocha" as "Whip, Double Mocha." Note that `getDescription()` could return an `ArrayList` of descriptions to make this easier.

SHARPEN YOUR PENCIL

Our friends at Starbuzz have introduced sizes to their menu. You can now order a coffee in tall, grande, and venti sizes (translation: small, medium, and large). Starbuzz saw this as an intrinsic part of the coffee class, so they've added two methods to the Beverage class: `setSize()` and `getSize()`. They'd also like for the condiments to be charged according to size, so for instance, Soy costs 10¢, 15¢, and 20¢, respectively, for tall, grande, and venti coffees. The updated Beverage class is shown below.

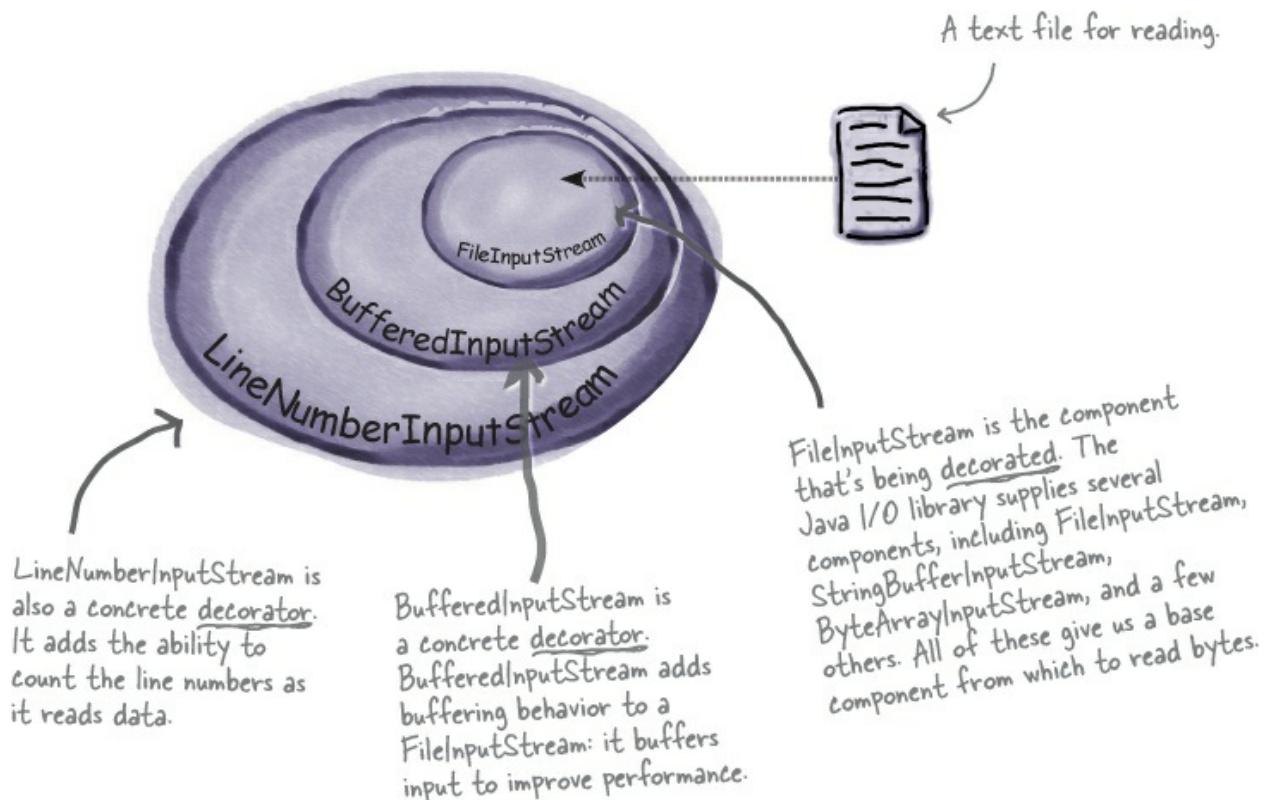
How would you alter the decorator classes to handle this change in requirements?

```
public abstract class Beverage {
    public enum Size { TALL, GRANDE, VENTI };
    Size size = Size.TALL;
    String description = "Unknown Beverage";
    public String getDescription() {
        return description;
    }
    public void setSize(Size size) {
        this.size = size;
    }
    public Size getSize() {
        return this.size;
    }
    public abstract double cost();
}
```

Real World Decorators: Java I/O

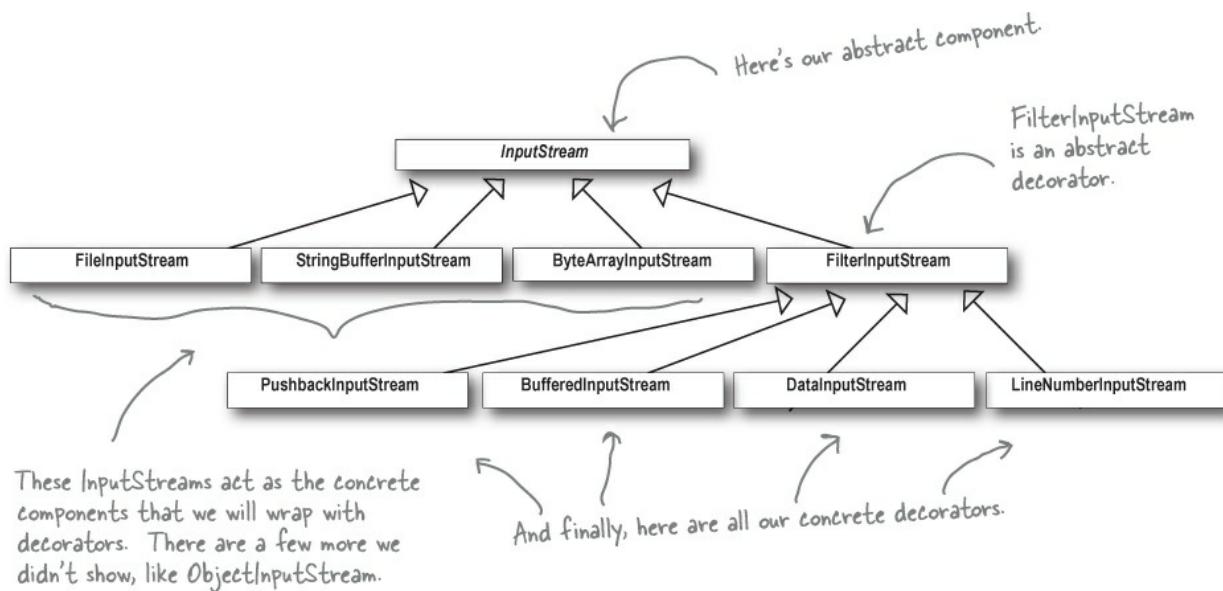
The large number of classes in the `java.io` package is... *overwhelming*. Don't

feel alone if you said “whoa” the first (and second and third) time you looked at this API. But now that you know the Decorator Pattern, the I/O classes should make more sense since the `java.io` package is largely based on Decorator. Here’s a typical set of objects that use decorators to add functionality to reading data from a file:



BufferedInputStream and **LineNumberInputStream** both extend **FilterInputStream**, which acts as the abstract decorator class.

Decorating the `java.io` classes



You can see that this isn't so different from the Starbuzz design. You should now be in a good position to look over the java.io API docs and compose decorators on the various *input* streams.

You'll see that the *output* streams have the same design. And you've probably already found that the Reader/Writer streams (for character-based data) closely mirror the design of the streams classes (with a few differences and inconsistencies, but close enough to figure out what's going on).

Java I/O also points out one of the *downsides* of the Decorator Pattern: designs using this pattern often result in a large number of small classes that can be overwhelming to a developer trying to use the Decorator-based API. But now that you know how Decorator works, you can keep things in perspective and when you're using someone else's Decorator-heavy API, you can work through how their classes are organized so that you can easily use wrapping to get the behavior you're after.

Writing your own Java I/O Decorator

Okay, you know the Decorator Pattern, you've seen the I/O class diagram. You should be ready to write your own input decorator.

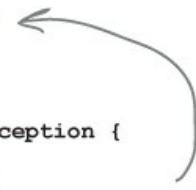
How about this: write a decorator that converts all uppercase characters to lowercase in the input stream. In other words, if we read in "I know the Decorator Pattern therefore I RULE!" then your decorator converts this to "i know the decorator pattern therefore i rule!"

Don't forget to import
java.io... (not shown).

First, extend the FilterInputStream, the
abstract decorator for all InputStreams.



```
public class LowerCaseInputStream extends FilterInputStream {  
  
    public LowerCaseInputStream(InputStream in) {  
        super(in);  
    }  
  
    public int read() throws IOException {  
        int c = in.read();  
        return (c == -1 ? c : Character.toLowerCase((char)c));  
    }  
  
    public int read(byte[] b, int offset, int len) throws IOException {  
        int result = in.read(b, offset, len);  
        for (int i = offset; i < offset+result; i++) {  
            b[i] = (byte)Character.toLowerCase((char)b[i]);  
        }  
        return result;  
    }  
}
```



Now we need to implement two
read methods. They take a
byte (or an array of bytes)
and convert each byte (that
represents a character) to
lowercase if it's an uppercase
character.



No problem. I
just have to extend the
FilterInputStream class and
override the read() methods.

REMEMBER: we don't provide import and package statements in the code listings.
Get the complete source code from <http://wickedlysmart.com/head-first-design-patterns/>.

Test out your new Java I/O Decorator

Write some quick code to test the I/O decorator:

```

public class InputTest {
    public static void main(String[] args) throws IOException {
        int c;
        try {
            InputStream in =
                new LowerCaseInputStream(
                    new BufferedInputStream(
                        new FileInputStream("test.txt")));
            while((c = in.read()) >= 0) {
                System.out.print((char)c);
            }
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Set up the FileInputStream and decorate it, first with a BufferedInputStream and then our brand new LowerCaseInputStream filter.

I know the Decorator Pattern therefore I RULE!

Just use the stream to read characters until the end of file and print as we go.

test.txt file

You need to make this file.

Give it a spin

```

File Edit Window Help DecoratorsRule
% java InputTest
i know the decorator pattern therefore i rule!
%

```

PATTERNS EXPOSED

This week's interview: Confessions of a Decorator

Head First: Welcome, Decorator Pattern. We've heard that you've been a bit down on yourself lately?

Decorator: Yes, I know the world sees me as the glamorous design pattern, but you know, I've got my share of problems just like everyone.

HeadFirst: Can you perhaps share some of your troubles with us?

Decorator: Sure. Well, you know I've got the power to add flexibility to designs, that much is for sure, but I also have a *dark side*. You see, I can sometimes add a lot of small

classes to a design and this occasionally results in a design that's less than straightforward for others to understand.

HeadFirst: Can you give us an example?

Decorator: Take the Java I/O libraries. These are notoriously difficult for people to understand at first. But if they just saw the classes as a set of wrappers around an `InputStream`, life would be much easier.

HeadFirst: That doesn't sound so bad. You're still a great pattern, and improving this is just a matter of public education, right?

Decorator: There's more, I'm afraid. I've got typing problems: you see, people sometimes take a piece of client code that relies on specific types and introduce decorators without thinking through everything. Now, one great thing about me is that ***you can usually insert decorators transparently and the client never has to know it's dealing with a decorator.*** But like I said, some code is dependent on specific types and when you start introducing decorators, boom! Bad things happen.

HeadFirst: Well, I think everyone understands that you have to be careful when inserting decorators. I don't think this is a reason to be too down on yourself.

Decorator: I know, I try not to be. I also have the problem that introducing decorators can increase the complexity of the code needed to instantiate the component. Once you've got decorators, you've got to not only instantiate the component, but also wrap it with who knows how many decorators.

HeadFirst: I'll be interviewing the Factory and Builder patterns next week — I hear they can be very helpful with this?

Decorator: That's true; I should talk to those guys more often.

HeadFirst: Well, we all think you're a great pattern for creating flexible designs and staying true to the Open-Closed Principle, so keep your chin up and think positively!

Decorator: I'll do my best, thank you.

Tools for your Design Toolbox

You've got another chapter under your belt and a new principle and pattern in the toolbox.

OO Basics

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

ction
ulation
rphism
iance

We now have the Open-Closed Principle to guide us. We're going to strive to design our system so that the closed parts are isolated from our new extensions.

OO Patterns

Strategic
encapsulation
interface
varying
data

Decorator - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

And here's our first pattern for creating designs that satisfy the Open-Closed Principle. Or was it really the first? Is there another pattern we've used that follows this principle as well?

BULLET POINTS

- Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in our designs.
- In our designs we should allow behavior to be extended without the need to modify existing code.
- Composition and delegation can often be used to add new behaviors at runtime.
- The Decorator Pattern provides an alternative to subclassing for extending behavior.
- The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components.
- Decorator classes mirror the type of the components they decorate. (In fact, they are the same type as the components they decorate, either through inheritance or interface implementation.)
- Decorators change the behavior of their components by adding new functionality before and/or after (or even in place of) method calls to the component.
- You can wrap a component with any number of decorators.
- Decorators are typically transparent to the client of the component; that is, unless the client is relying on the component's concrete type.
- Decorators can result in many small objects in our design, and overuse can be complex.

SHARPEN YOUR PENCIL SOLUTION

Write the cost() methods for the following classes (pseudo-Java is okay). Here's our solution:

```
public class Beverage {  
  
    // declare instance variables for milkCost,  
    // soyCost, mochaCost, and whipCost, and  
    // getters and setters for milk, soy, mocha  
    // and whip.  
  
    public double cost() {  
  
        float condimentCost = 0.0;  
        if (hasMilk()) {  
            condimentCost += milkCost;  
        }  
        if (hasSoy()) {  
            condimentCost += soyCost;  
        }  
        if (hasMocha()) {  
            condimentCost += mochaCost;  
        }  
        if (hasWhip()) {  
            condimentCost += whipCost;  
        }  
        return condimentCost;  
    }  
}
```

```

}

public class DarkRoast extends Beverage {

    public DarkRoast() {
        description = "Most Excellent Dark Roast";
    }

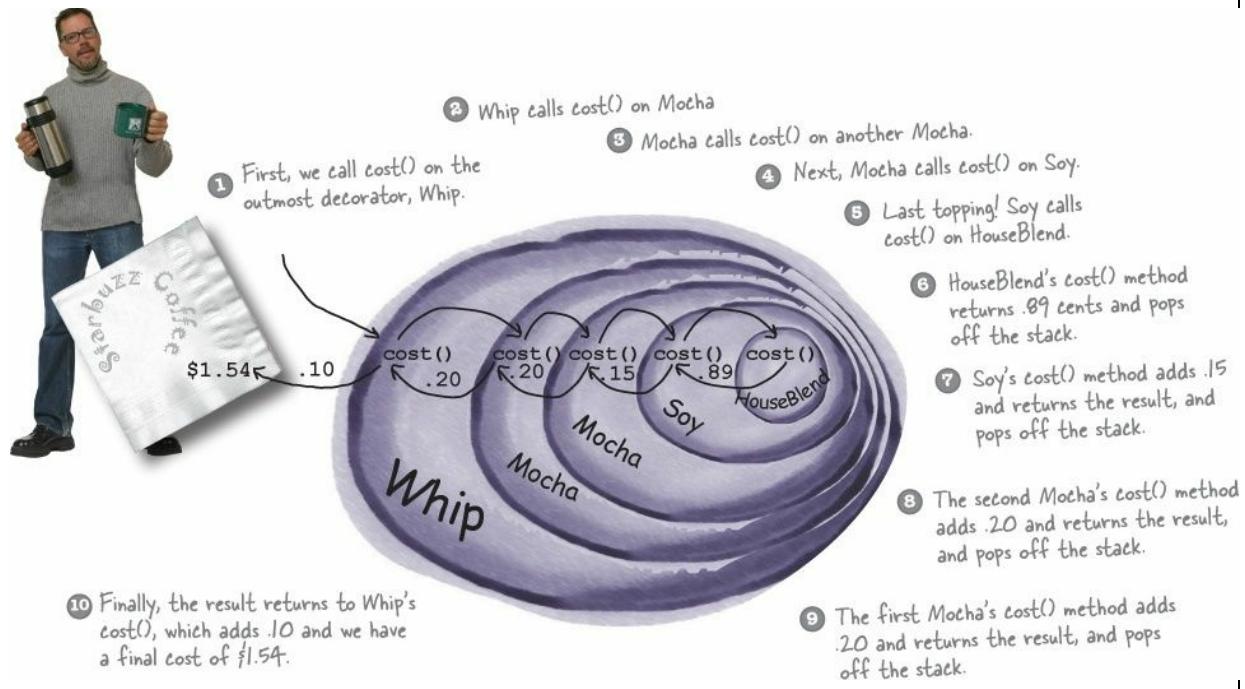
    public double cost() {
        return 1.99 + super.cost();
    }
}

```

SHARPEN YOUR PENCIL SOLUTION

New barista training

“double mocha soy latte with whip”



SHARPEN YOUR PENCIL SOLUTION

Our friends at Starbuzz have introduced sizes to their menu. You can now order a coffee in tall, grande, and venti sizes (for us normal folk: small, medium, and large). Starbuzz saw this as an intrinsic part of the coffee class, so they've added two methods to the Beverage class: `setSize()` and `getSize()`. They'd also like for the condiments to be charged according to size, so for instance, Soy costs 10¢, 15¢, and 20¢, respectively, for tall, grande, and venti coffees.

How would you alter the decorator classes to handle this change in requirements? Here's

our solution.

```
public abstract class CondimentDecorator extends Beverage {  
    public Beverage beverage;  
    public abstract String getDescription();  
  
    public Size getSize() {  
        return beverage.getSize();  
    }  
}  
  
public class Soy extends CondimentDecorator {  
    public Soy(Beverage beverage) {  
        this.beverage = beverage;  
    }  
  
    public String getDescription() {  
        return beverage.getDescription() + ", Soy";  
    }  
  
    public double cost() {  
        double cost = beverage.cost();  
        if (beverage.getSize() == Size.TALL) {  
            cost += .10;  
        } else if (beverage.getSize() == Size.GRANDE) {  
            cost += .15;  
        } else if (beverage.getSize() == Size.VENTI) {  
            cost += .20;  
        }  
        return cost;  
    }  
}
```

We moved the Beverage instance variable into CondimentDecorator, and added a method, getSize() for the decorators that simply returns the size of the beverage.

Here we get the size (which propagates all the way to the concrete beverage) and then add the appropriate cost.

Chapter 4. The Factory Pattern: Baking with OO Goodness



Get ready to bake some loosely coupled OO designs. There is more to making objects than just using the **new** operator. You'll learn that instantiation is an activity that shouldn't always be done in public and can often lead to *coupling problems*. And you don't want *that*, do you? Find out how Factory Patterns can help save you from embarrassing dependencies.



When you see “new,” think “concrete.”

Yes, when you use `new` you are certainly instantiating a concrete class, so that's definitely an implementation, not an interface. And it's a good question; you've learned that tying your code to a concrete class can make it more fragile and less flexible.

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

We want to use interfaces
to keep code flexible.

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

When you have a whole set of related concrete classes, often you're forced to write code like this:

```
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.

Here we've got several concrete classes being instantiated, and the decision of which to instantiate is made at runtime depending on some set of conditions.

When you see code like this, you know that when it comes time for changes or extensions, you'll have to reopen this code and examine what needs to be added (or deleted). Often this kind of code ends up in several parts of the application making maintenance and updates more difficult and error-prone.

But you have to create an object at some point and Java only gives us one way to create an object, right? So what gives?



What's wrong with “new”?

Technically there's nothing wrong with **new**. After all, it's a fundamental part of Java. The real culprit is our old friend CHANGE and how change impacts our use of **new**.

By coding to an interface, you know you can insulate yourself from a lot of changes that might happen to a system down the road. Why? If your code is written to an interface, then it will work with any new classes implementing that interface through polymorphism. However, 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. So, in other words, your code will not be "closed for modification." To extend it with new concrete types, you'll have to reopen it.

NOTE

Remember that designs should be "open for extension but closed for modification" - see [Chapter 3](#) for a review.

So what can you do? It's times like these that you can fall back on OO Design Principles to look for clues. Remember, our first principle deals with change and guides us to *identify the aspects that vary and separate them from what stays the same*.

BRAIN POWER

How might you take all the parts of your application that instantiate concrete classes and separate or encapsulate them from the rest of your application?

Identifying the aspects that vary



Let's say you have a pizza shop, and as a cutting-edge pizza store owner in Objectville you might end up writing some code like this:

```
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.

But you need more than one type of pizza...

So then you'd add some code that *determines* the appropriate type of pizza and then goes about *making* the pizza:

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

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

```
if (type.equals("cheese")) {  
    pizza = new CheesePizza();  
} else if (type.equals("greek")) {  
    pizza = new GreekPizza();  
} else if (type.equals("pepperoni")) {  
    pizza = new PepperoniPizza();  
}
```

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.

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

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.

But the pressure is on to add more pizza types

You realize that all of your competitors have added a couple of trendy pizzas

to their menus: the Clam Pizza and the Veggie Pizza. Obviously you need to keep up with the competition, so you'll add these items to your menu. And you haven't been selling many Greek Pizzas lately, so you decide to take that off the menu:

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

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.

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.

Encapsulating object creation

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.

```

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.

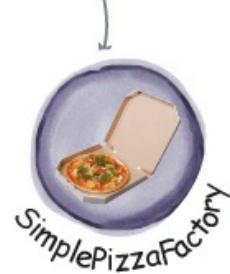
What's going to go here?

```

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();
}

```

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.



We've got a name for this new object: we call it a Factory.

Factories handle the details of object creation. Once we have a SimplePizzaFactory, our orderPizza() method just becomes a client of that object. Any time it needs a pizza it asks the pizza factory to make one. Gone are the days when the orderPizza() method needs to know about Greek versus Clam pizzas. Now the orderPizza() method just cares that it gets a pizza that implements the Pizza interface so that it can call prepare(), bake(), cut(), and box().

We've still got a few details to fill in here; for instance, what does the orderPizza() method replace its creation code with? Let's implement a simple factory for the pizza store and find out...

Building a simple pizza factory

We'll start with the factory itself. What we're going to do is define a class that encapsulates the object creation for all pizzas. Here it is...

Here's our new class, the SimplePizzaFactory. It has one job in life: creating pizzas for its clients.

```
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;  
    }  
}
```

First we define a createPizza() method in the factory. This is the method all clients will use to instantiate new objects.

Here's the code we plucked out of the orderPizza() method.

This code is still parameterized by the type of the pizza, just like our original orderPizza() method was.

THERE ARE NO DUMB QUESTIONS

Q: Q: What's the advantage of this? It looks like we are just pushing the problem off to another object.

A: A: One thing to remember is that the SimplePizzaFactory may have many clients. We've only seen the orderPizza() method; however, there may be a PizzaShopMenu class that uses the factory to get pizzas for their current description and price. We might also have a HomeDelivery class that handles pizzas in a different way than our PizzaShop class but is also a client of the factory.

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

Don't forget, we are also just about to remove the concrete instantiations from our client code.

Q: Q: I've seen a similar design where a factory like this is defined as a static method. What is the difference?

A: A: Defining a simple factory as a static method is a common technique and is often called a static factory. Why use a static method? Because you don't need to instantiate an object to make use of the create method. But remember it also has the disadvantage that you can't subclass and change the behavior of the create method.

Reworking the PizzaStore class

Now it's time to fix up our client code. What we want to do is rely on the factory to create the pizzas for us. Here are the changes:

```
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!

BRAIN POWER

Q: We know that object composition allows us to change behavior dynamically at runtime (among other things) because we can swap in and out implementations. How might we be able to use that in the PizzaStore? What factory implementations might we be able to swap in and out?

A: We don't know about you, but we're thinking New York, Chicago, and California style pizza factories (let's not forget New Haven, too)

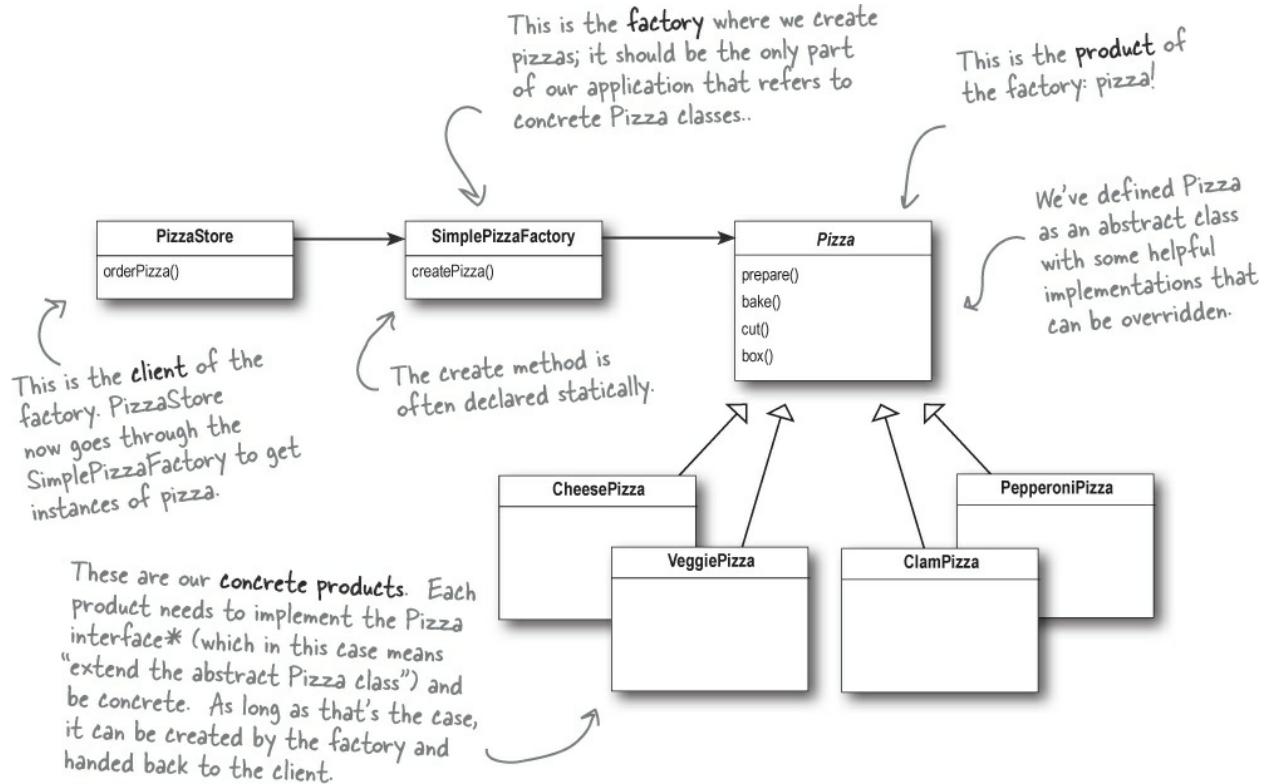
The Simple Factory defined



Pattern Honorable Mention

The Simple Factory isn't actually a Design Pattern; it's more of a programming idiom. But it is commonly used, so we'll give it a Head First Pattern Honorable Mention. Some developers do mistake this idiom for the "Factory Pattern," so the next time there is an awkward silence between you and another developer, you've got a nice topic to break the ice.

Just because Simple Factory isn't a REAL pattern doesn't mean we shouldn't check out how it's put together. Let's take a look at the class diagram of our new Pizza Store:



Think of Simple Factory as a warm up. Next, we'll explore two heavy-duty patterns that are both factories. But don't worry, there's more pizza to come!

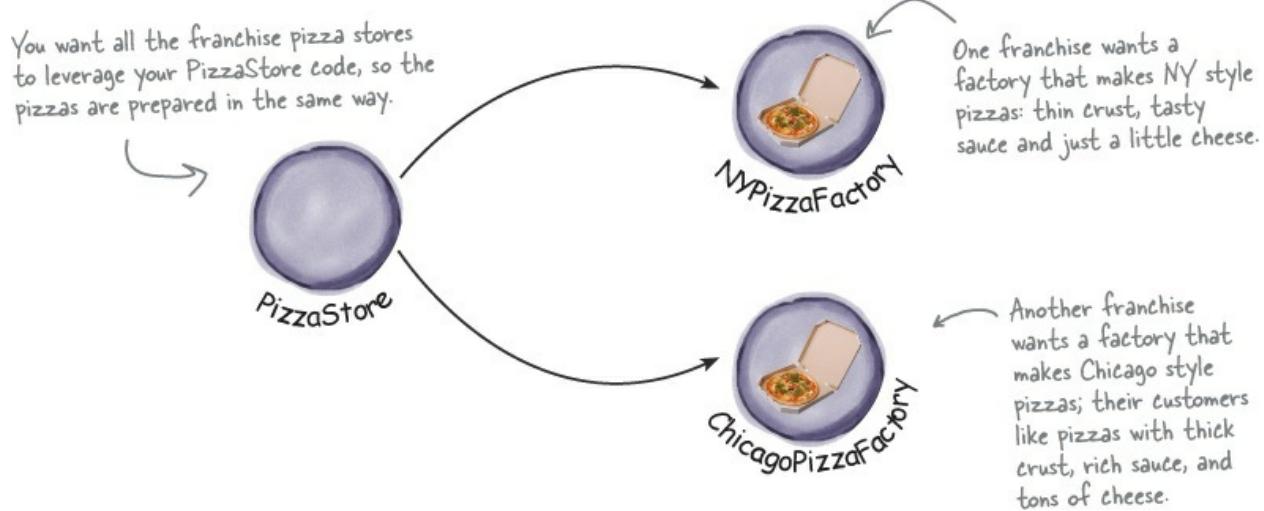
NOTE

*Just another reminder: in design patterns, the phrase “implement an interface” does NOT always mean “write a class that implements a Java interface, by using the ‘implements’ keyword in the class declaration.” In the general use of the phrase, a concrete class implementing a method from a supertype (which could be a class OR interface) is still considered to be “implementing the interface” of that supertype.

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.orderPizza("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 style pizzas.

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

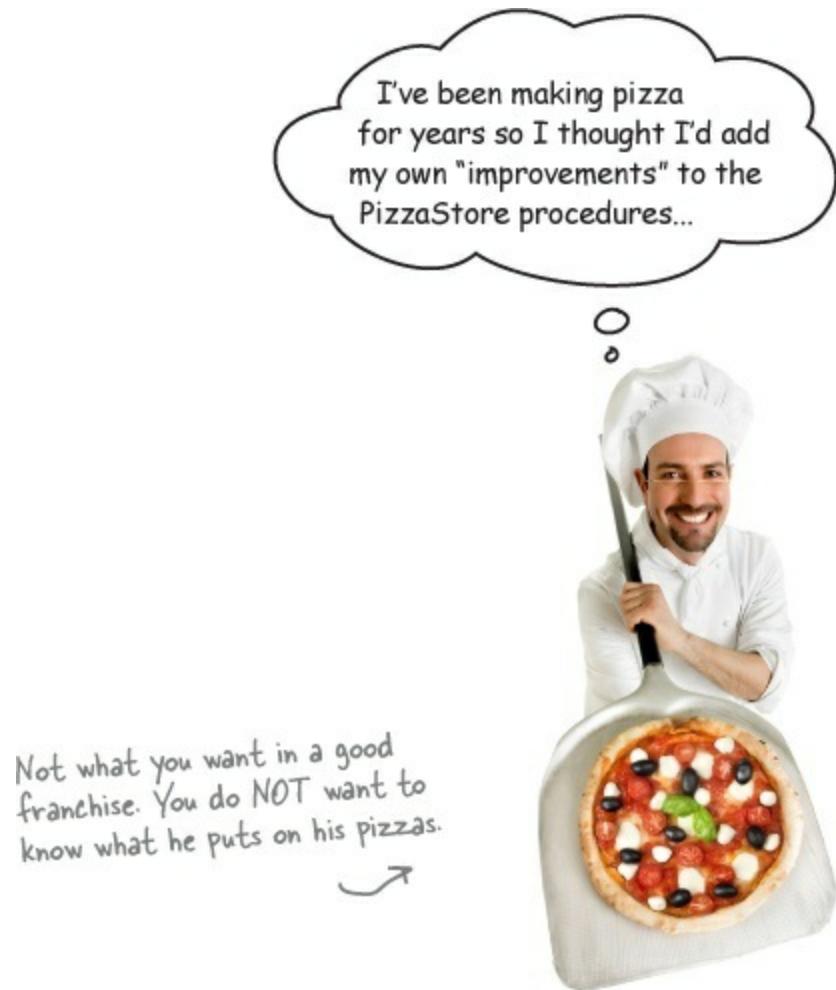
Likewise for the Chicago pizza stores: we
create a factory for Chicago pizzas and
create a store that is composed with a
Chicago factory. When we make pizzas, we
get the Chicago style ones.

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?



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.

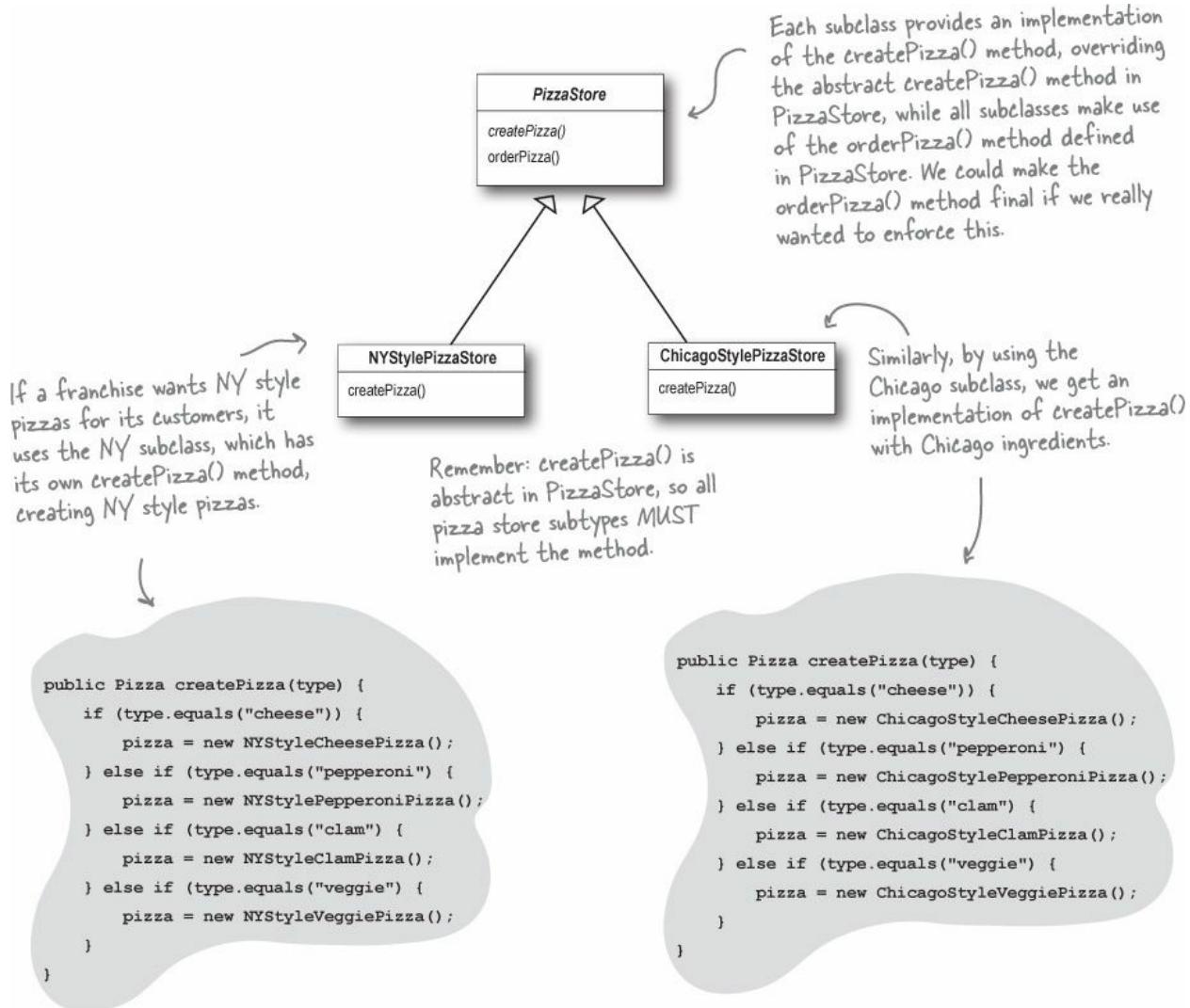
Now we've got a store waiting for subclasses; we're going to have a subclass for each regional type (NYPizzaStore, ChicagoPizzaStore, CaliforniaPizzaStore) and each subclass is going to make the decision about what makes up a pizza. Let's take a look at how this is going to work.

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

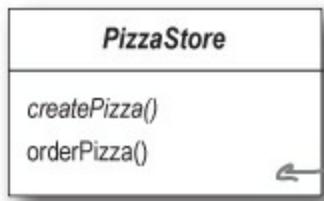
make it responsible for creating the right kind of pizza. The way we do this is by letting each subclass of PizzaStore define what the `createPizza()` method looks like. So, we will have a number of concrete subclasses of PizzaStore, each with its own pizza variations, all fitting within the PizzaStore framework and still making use of the well-tuned `orderPizza()` method.





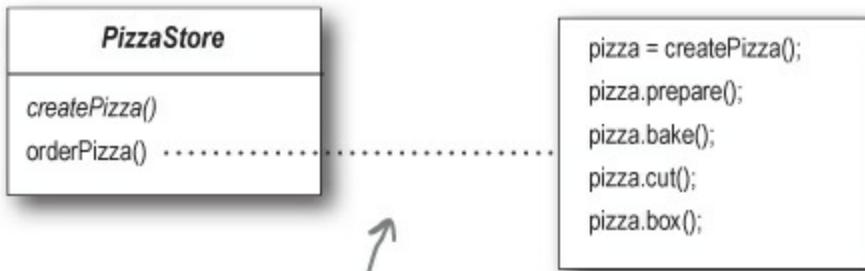
I don't get it. The PizzaStore subclasses are just subclasses. How are they deciding anything? I don't see any logical decision-making code in NYStylePizzaStore....

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.



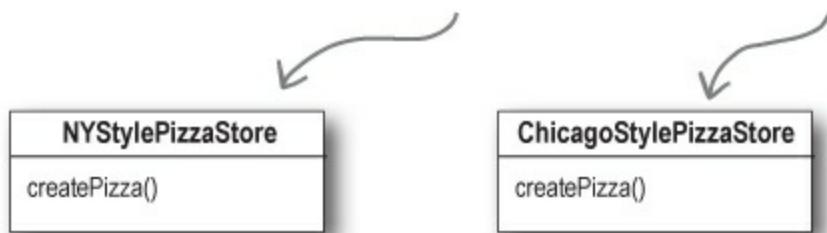
orderPizza() is defined in the abstract PizzaStore, not the subclasses. So, the method has no idea which subclass is actually running the code and making the pizzas.

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!



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?

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

* 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 NY Pizza Store; 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);

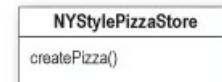
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        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.

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

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 a 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...



Behind the Scenes

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).

nyPizzaStore

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.



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();
```

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

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

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.

```

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;
    }
}

```

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.

NOTE

REMEMBER: we don't provide import and package statements in the code listings. Get the complete source code from the wickedlysmart website. 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.

```
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");  
    }  
}
```

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.

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 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 Reggiano 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

```

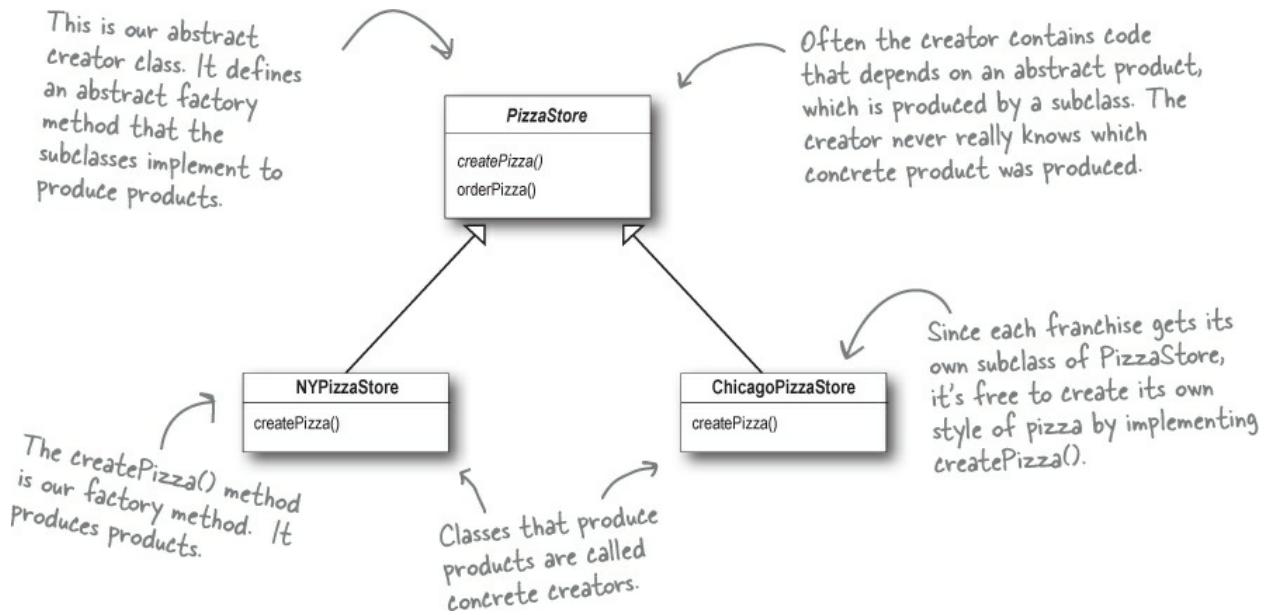
Both pizzas get prepared, the toppings added, and the pizzas baked, cut and boxed. Our superclass never had to know the details, the subclass handled all that just by instantiating the right pizza.

It's finally time to meet the Factory Method Pattern

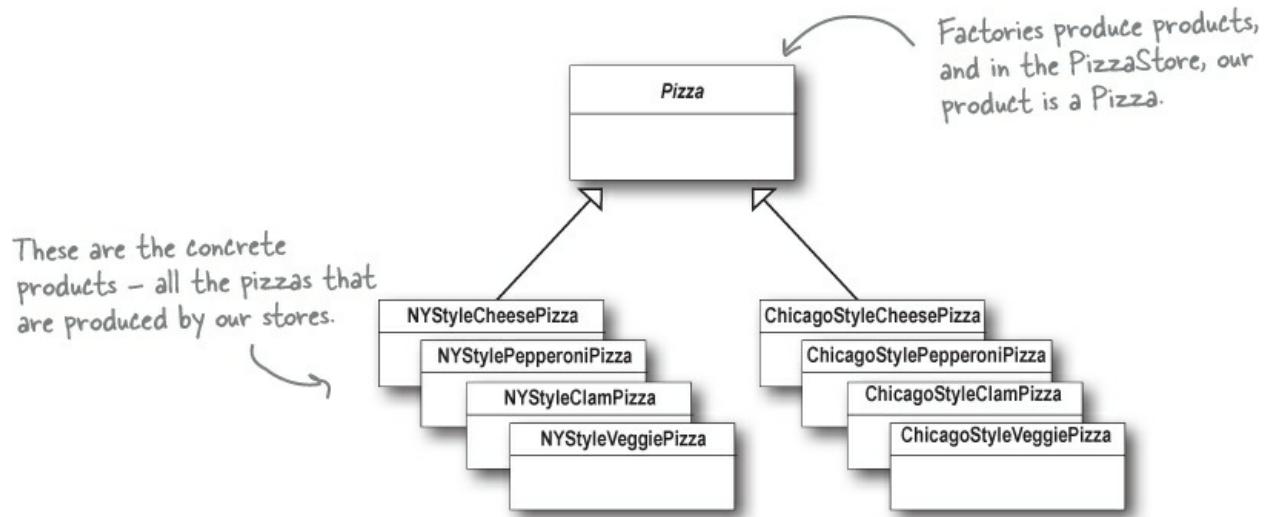
All factory patterns encapsulate object creation. The Factory Method Pattern encapsulates object creation by letting subclasses decide what objects to create. Let's check out these class diagrams to see who the players are in this

pattern:

The Creator classes



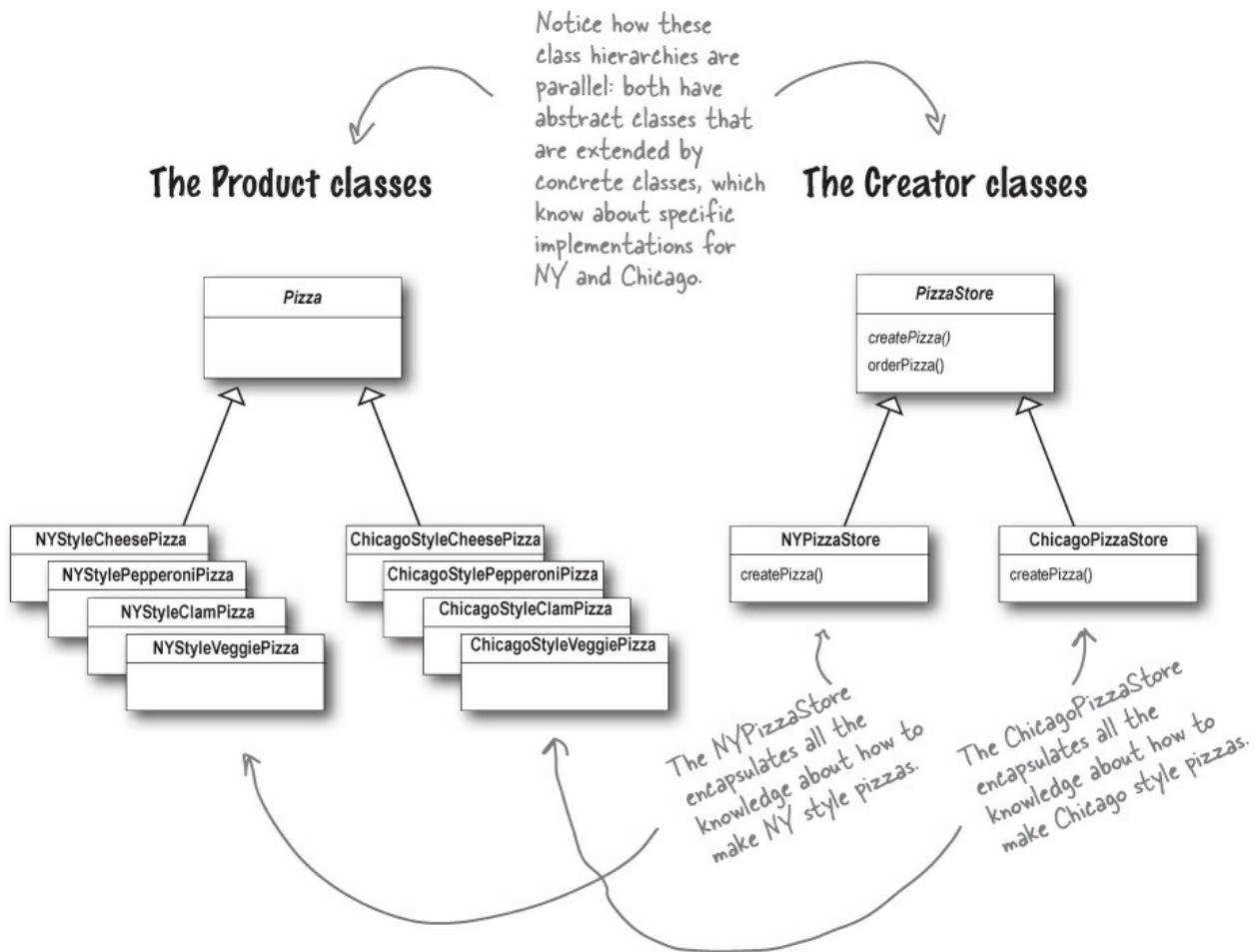
The Product classes



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:

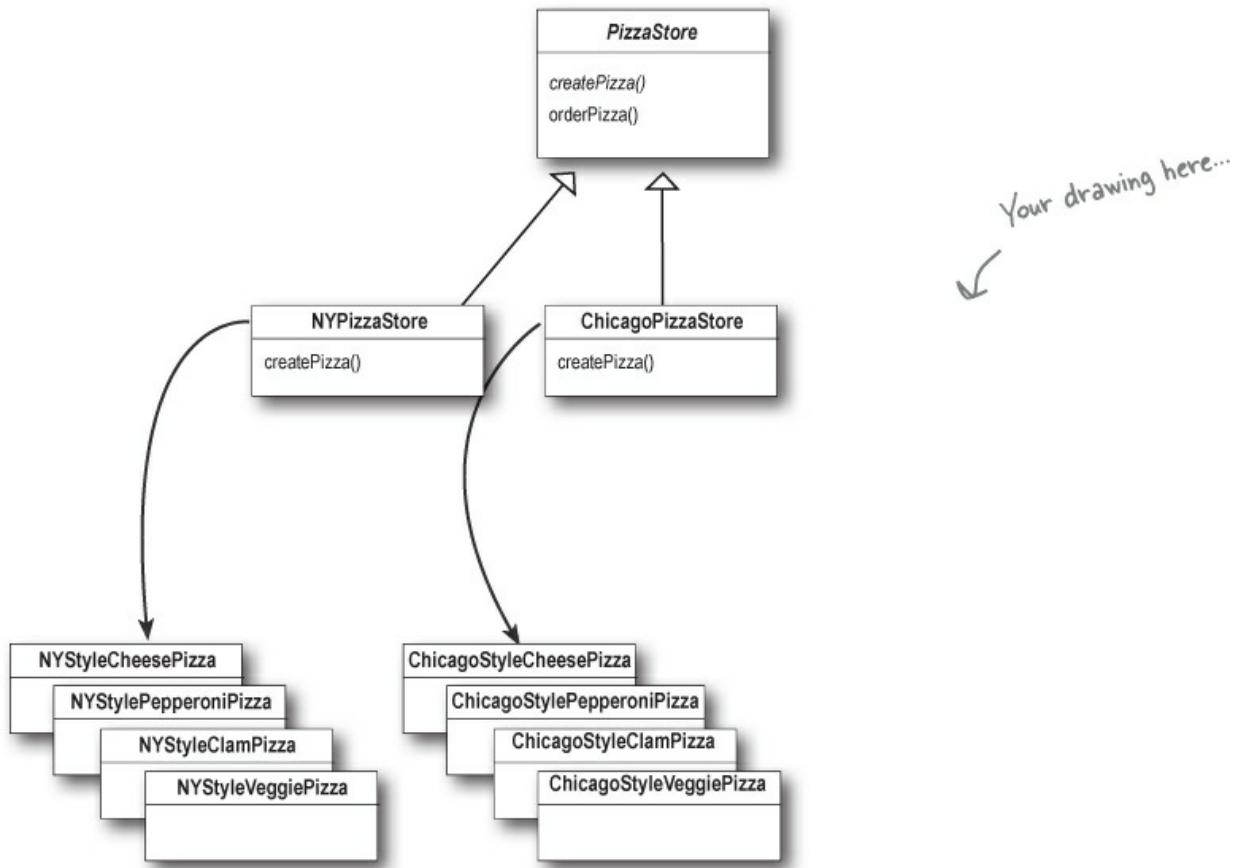


NOTE

The factory method is the key to encapsulating this knowledge.

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.



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:

NOTE

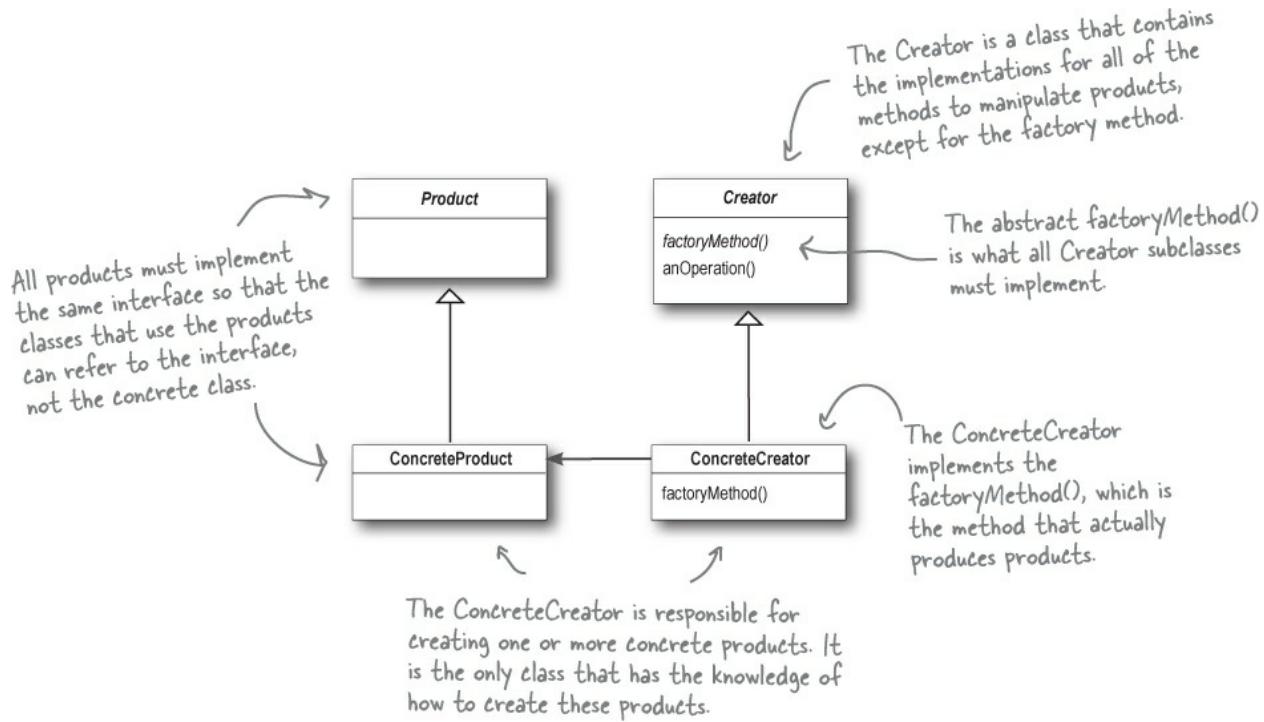
The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

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 “decide” 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.

NOTE

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



THERE ARE NO DUMB QUESTIONS

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

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

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

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

A: 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 use enums.

Q: 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: 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 lets 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.

A very dependent PizzaStore

SHARPEN YOUR PENCIL

Let’s pretend you’ve never heard of an OO factory. Here’s a version of the PizzaStore that doesn’t use a factory; make a count of the number of concrete pizza objects this class is dependent on. If you added California style pizzas to this PizzaStore, how many objects would it be dependent on then?

```

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;
    }
}

```

You can write your
answers here:

number _____

Handles all the
NY style pizzas

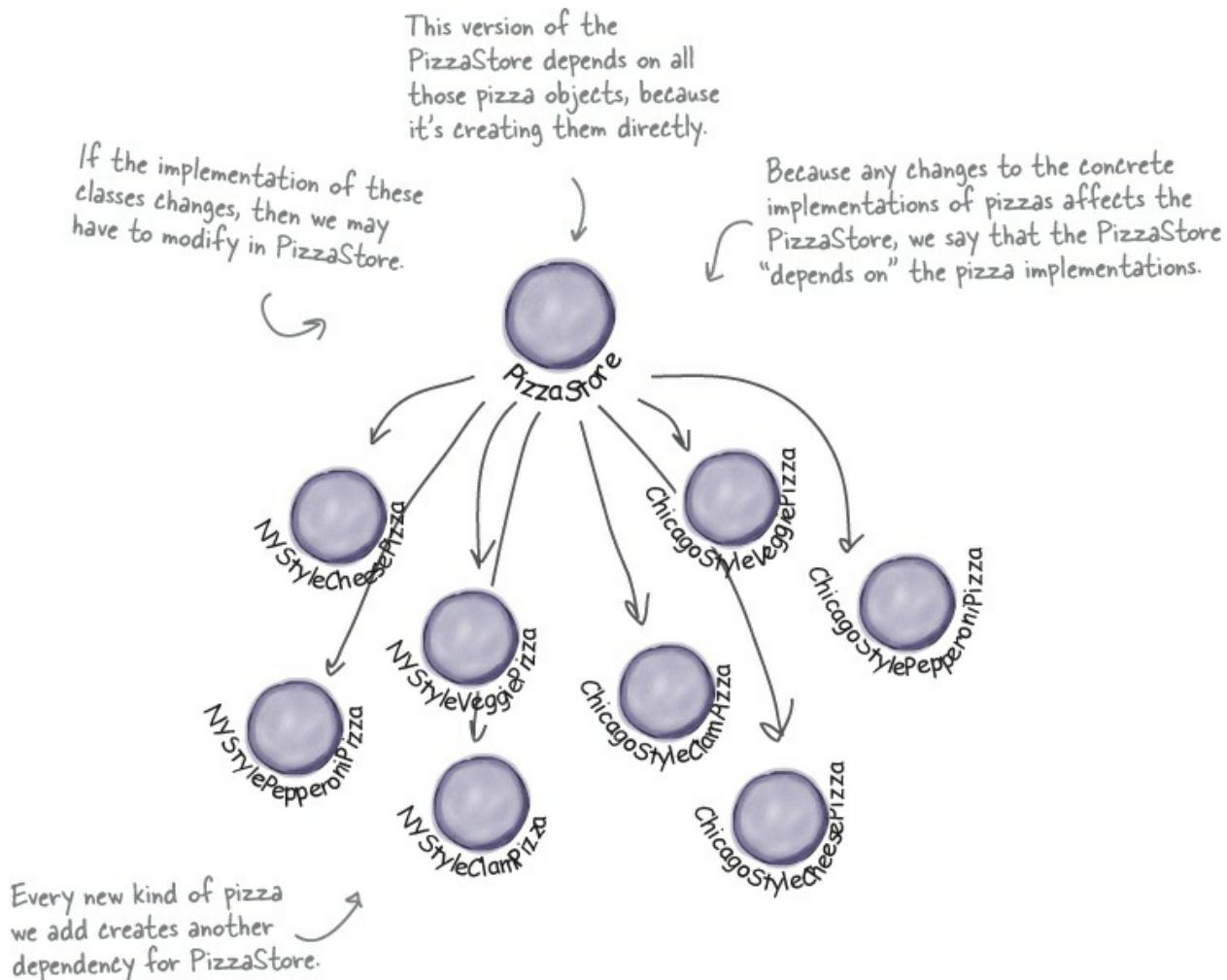
Handles all the
Chicago style pizzas

number with California too

Looking at object dependencies

When you directly instantiate an object, you are depending on its concrete class. Take a look at our very dependent PizzaStore one page back. It creates all the pizza objects right in the PizzaStore class instead of delegating to a factory.

If we draw a diagram representing that version of the PizzaStore and all the objects it depends on, here's what it looks like:



The Dependency Inversion Principle

It should be pretty clear that reducing dependencies to concrete classes in our code is a “good thing.” In fact, we’ve got an OO design principle that formalizes this notion; it even has a big, formal name: *Dependency Inversion Principle*.

NOTE

Yet another phrase you can use to impress the execs in the room! Your raise will more than offset the cost of this book, and you’ll gain the admiration of your fellow developers.

Here’s the general principle:

DESIGN PRINCIPLE

Depend upon abstractions. Do not depend upon concrete classes.

At first, this principle sounds a lot like “Program to an interface, not an implementation,” right? It is similar; however, the Dependency Inversion Principle makes an even stronger statement about abstraction. It suggests that our high-level components should not depend on our low-level components; rather, they should *both* depend on abstractions.

NOTE

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, and prepares, bakes, cuts, and boxes them, while the pizzas it uses are low-level components.

But what the heck does that mean?

Well, let’s start by looking again at the pizza store diagram on the previous page. PizzaStore is our “high-level component” and the pizza implementations are our “low-level components,” and clearly the PizzaStore is dependent on the concrete pizza classes.

Now, this principle tells us we should instead write our code so that we are depending on abstractions, not concrete classes. That goes for both our high-level modules and our low-level modules.

But how do we do this? Let’s think about how we’d apply this principle to our Very Dependent PizzaStore implementation...

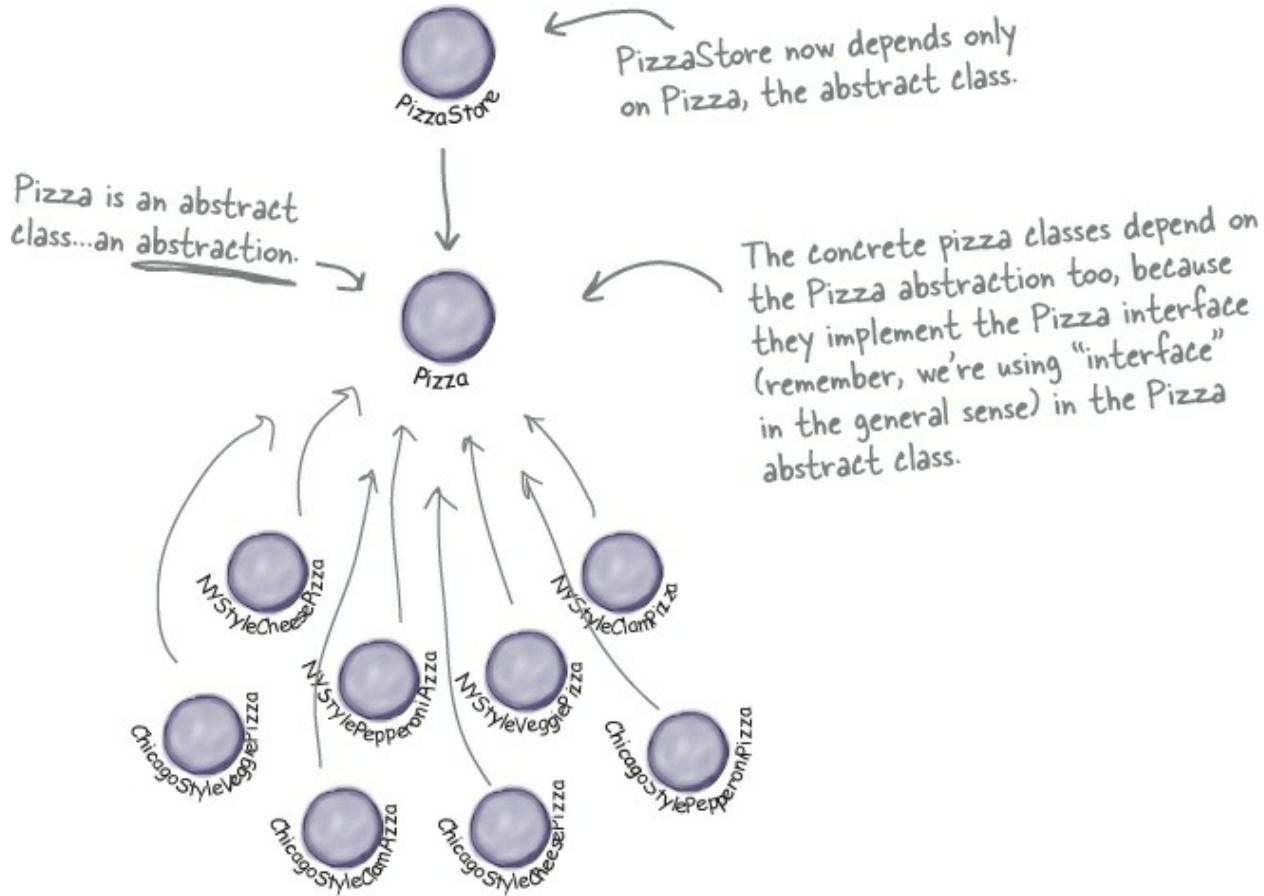
Applying the Principle

Now, the main problem with the Very Dependent PizzaStore is that it depends on every type of pizza because it actually instantiates concrete types in its `orderPizza()` method.

While we’ve created an abstraction, `Pizza`, we’re nevertheless creating concrete `Pizzas` in this code, so we don’t get a lot of leverage out of this abstraction.

How can we get those instantiations out of the `orderPizza()` method? Well, as we know, the Factory Method allows us to do just that.

So, after we've applied the Factory Method, our diagram looks like this:



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.



Where's the “inversion” in Dependency Inversion Principle?

The “inversion” in the name Dependency Inversion Principle is there because it inverts the way you typically might think about your OO design. Look at the diagram on the previous page. Notice that the low-level components now depend on a higher level abstraction. Likewise, the high-level component is also tied to the same abstraction. So, the top-to-bottom dependency chart we drew a couple of pages back has inverted itself, with both high-level and low-level modules now depending on the abstraction.

Let's also walk through the thinking behind the typical design process and see how introducing the principle can invert the way we think about the design...

Inverting your thinking...



	Okay, so you need to implement a PizzaStore. What's the first thought that pops into your head?
💡	<p>Right, you start at the top and follow things down to the concrete classes. But, as you've seen, you don't want your store to know about the concrete pizza types, because then it'll be dependent on all those concrete classes!</p> <p>Now, let's "invert" your thinking... instead of starting at the top, start at the Pizzas and think about what you can abstract.</p>
💡	<p>Right! You are thinking about the abstraction <i>Pizza</i>. So now, go back and think about the design of the Pizza Store again.</p> <p>Close. But to do that you'll have to rely on a factory to get those concrete classes out of your Pizza Store. Once you've done that, your different concrete pizza types depend only on an abstraction and so does your store. We've taken a design where the store depended on concrete classes and inverted those dependencies (along with your thinking).</p>

A few guidelines to help you follow the Principle...

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.

NOTE

If you use **new**, you'll be holding a reference to a concrete class. Use a factory to get around that!

- No class should derive from a concrete class.

NOTE

If you derive from a concrete class, you're depending on a concrete class. Derive from an abstraction, like an interface or an abstract class.

- No method should override an implemented method of any of its base classes.

NOTE

If you override an implemented method, then your base class wasn't really an

abstraction to start with. Those methods implemented in the base class are meant to be shared by all your subclasses.

You're exactly right! Like many of our principles, this is a guideline you should strive for, rather than a rule you should follow all the time. Clearly, every single Java program ever written violates these guidelines!

But, if you internalize these guidelines and have them in the back of your mind when you design, you'll know when you are violating the principle and you'll have a good reason for doing so. For instance, if you have a class that isn't likely to change, and you know it, then it's not the end of the world if you instantiate a concrete class in your code. Think about it; we instantiate String objects all the time without thinking twice. Does that violate the principle? Yes. Is that okay? Yes. Why? Because String is very unlikely to change.

If, on the other hand, a class you write is likely to change, you have some good techniques like Factory Method to encapsulate that change.



Meanwhile, back at the PizzaStore...

The design for the PizzaStore is really shaping up: it's got a flexible framework and it does a good job of adhering to design principles.

Now, the key to Objectville Pizza's success has always been fresh, quality ingredients, and what you've discovered is that with the new framework your franchises have been following your *procedures*, but a few franchises have been substituting inferior ingredients in their pies to lower costs and increase their margins. You know you've got to do something, because in the long term this is going to hurt the Objectville brand!



Ensuring consistency in your ingredients

So how are you going to ensure each franchise is using quality ingredients? You're going to build a factory that produces them and ships them to your franchises!

Now there is only one problem with this plan: the franchises are located in different regions and what is red sauce in New York is not red sauce in Chicago. So, you have one set of ingredients that needs to be shipped to New York and a *different* set that needs to be shipped to Chicago. Let's take a closer look:



Chicago Pizza Menu

Cheese Pizza

Plum Tomato Sauce, Mozzarella, Parmesan, Oregano

Veggie Pizza

Plum Tomato Sauce, Mozzarella, Parmesan, Eggplant, Spinach, Black Olives

Clam Pizza

Plum Tomato Sauce, Mozzarella, Parmesan, Clams

Pepperoni Pizza

Plum Tomato Sauce, Mozzarella, Parmesan, Eggplant, Spinach, Black Olives, Pepperoni

We've got the same product families (dough, sauce, cheese, veggies, meats) but different implementations based on region.



New York Pizza Menu

Cheese Pizza

Marinara Sauce, Reggiano, Garlic

Veggie Pizza

Marinara Sauce, Reggiano, Mushrooms, Onions, Red Peppers

Clam Pizza

Marinara Sauce, Reggiano, Fresh Clams

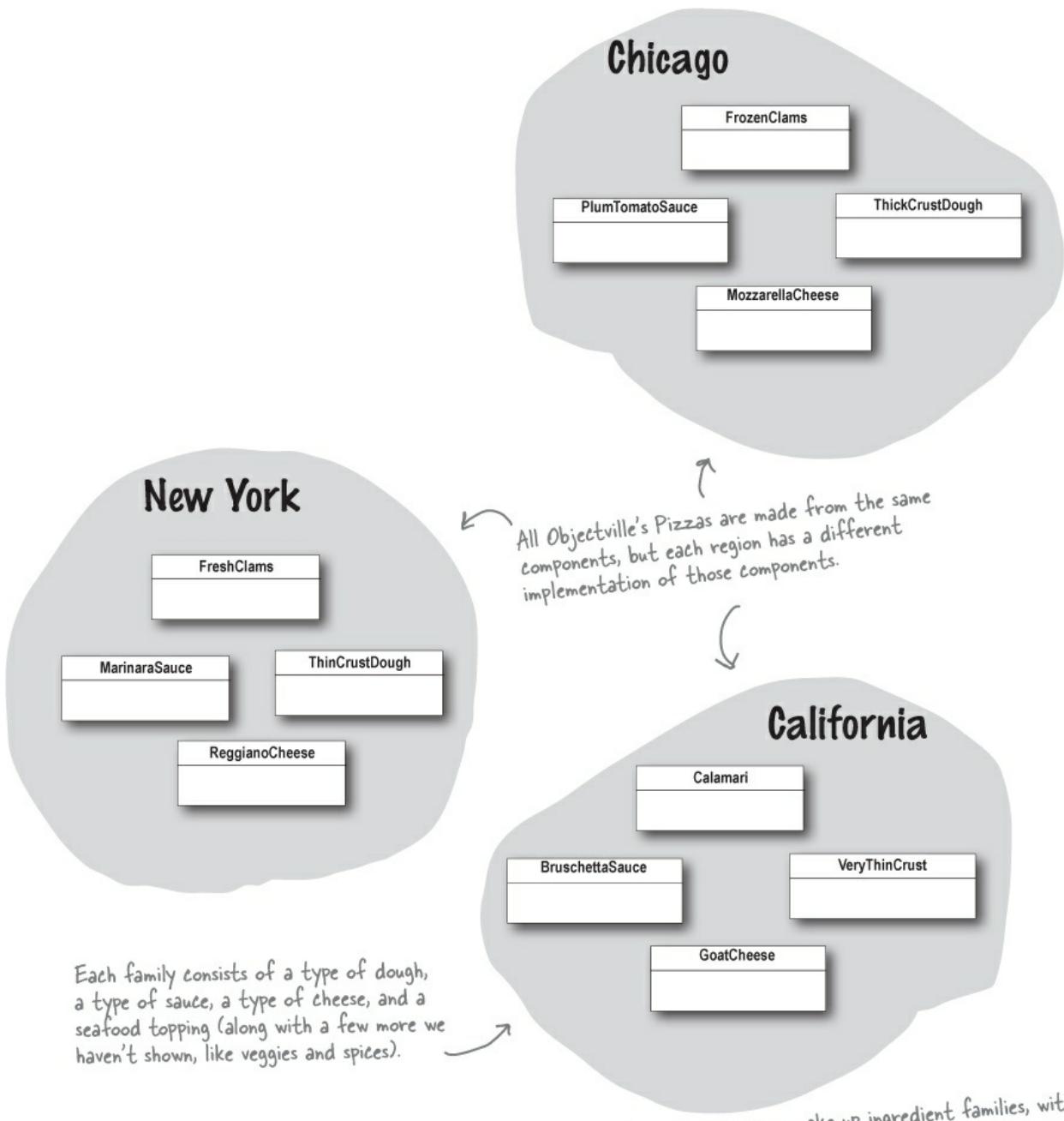
Pepperoni Pizza

Marinara Sauce, Reggiano, Mushrooms, Onions, Red Peppers, Pepperoni

Families of ingredients...

New York uses one set of ingredients and Chicago another. Given the popularity of Objectville Pizza, it won't be long before you also need to ship another set of regional ingredients to California, and what's next? Seattle?

For this to work, you are going to have to figure out how to handle families of ingredients.



Building the ingredient factories

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. In other words, the factory will need to create dough, sauce, cheese, and so on... You'll see how we are going to handle the regional differences shortly.

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();  
}
```

Lots of new classes here,
one per ingredient.



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

NOTE

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

Here's what we're going to do:

- ① Build a factory for each region. To do this, you'll create a subclass of `PizzaIngredientFactory` that implements each `create` method.
- ② 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.
- ③ Then we still need to hook all this up by working our new ingredient factories into our old `PizzaStore` code.

Building the New York ingredient factory

Okay, here's the implementation for the New York ingredient factory. This factory specializes in **Marinara Sauce, Reggiano Cheese, Fresh Clams...**

The NY ingredient factory implements the interface for all ingredient factories

```
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();  
    }  
}
```

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

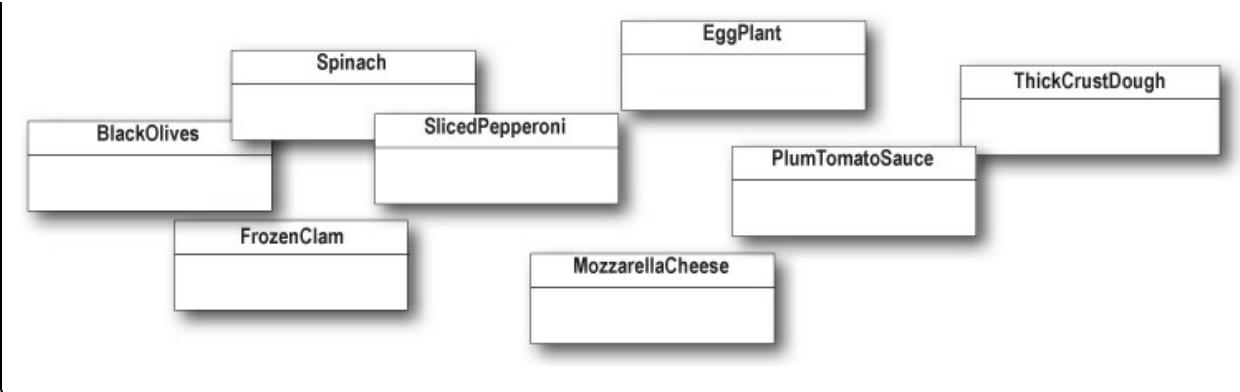
For each ingredient in the ingredient family, we create the New York version.

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

SHARPEN YOUR PENCIL

Write the ChicagoPizzaIngredientFactory. You can reference the classes below in your implementation:



Reworking the pizzas...

We've got our factories all fired up and ready to produce quality ingredients; now we just need to rework our Pizzas so they only use factory-produced ingredients. We'll start with our abstract 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");
    }
    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.

Reworking the pizzas, continued...

Now that you've got an abstract Pizza to work from, it's time to create the New York and Chicago style Pizzas — only this time around they will get their ingredients straight from the factory. The franchisees' days of skimping

on ingredients are over!

When we wrote the Factory Method code, we had a NYCheesePizza and a ChicagoCheesePizza class. If you look at the two classes, the only thing that differs is the use of regional ingredients. The pizzas are made just the same (dough + sauce + cheese). The same goes for the other pizzas: Veggie, Clam, and so on. They all follow the same preparation steps; they just have different ingredients.

So, what you'll see is that we really don't need two classes for each pizza; the ingredient factory is going to handle the regional differences for us. Here's the Cheese Pizza:

```
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.

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

← Here's where the magic happens!

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.

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.

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

Let's check out the ClamPizza as well:

```
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();  
    }  
}
```

ClamPizza also stashes
an ingredient factory.

To make a clam pizza, the prepare
method collects the right
ingredients from its local factory.

If it's a New York factory,
the clams will be fresh; if it's
Chicago, they'll be frozen.

T

Revisiting our pizza stores

We're almost there; we just need to make a quick trip to our franchise stores to make sure they are using the correct Pizzas. We also need to give them a reference to their local ingredient factories:

```

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.

BRAIN POWER

Compare this version of the createPizza() method to the one in the Factory Method implementation earlier in the chapter.

What have we done?

That was quite a series of code changes; what exactly did we do?

We provided a means of creating a family of ingredients for pizzas by introducing a new type of factory called an Abstract Factory.

An Abstract Factory gives us an interface for creating a family of products. By writing code that uses this interface, we decouple our code

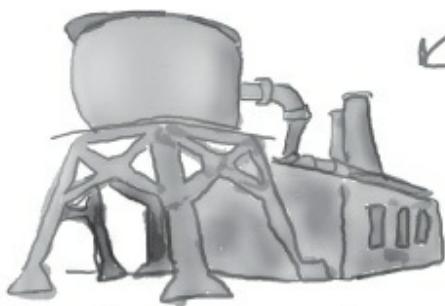
from the actual factory that creates the products. That allows us to implement a variety of factories that produce products meant for different contexts — such as different regions, different operating systems, or different look and feels.

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).

An Abstract Factory provides an interface for a family of products. What's a family? In our case, it's all the things we need to make a pizza: dough, sauce, cheese, meats, and veggies.

From the abstract factory, we derive one or more concrete factories that produce the same products, but with different implementations.

We then write our code so that it uses the factory to create products. By passing in a variety of factories, we get a variety of implementations of those products. But our client code stays the same.



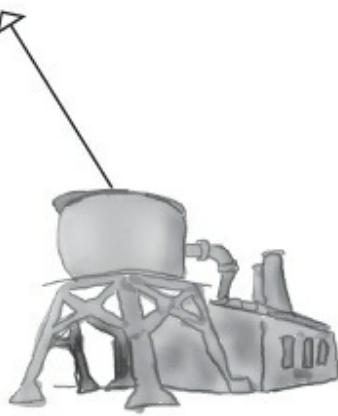
Defines the interface.

Objectville Abstract IngredientFactory

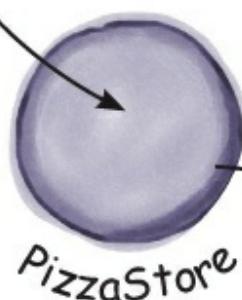
Provides implementations for products.



New York



Chicago



PizzaStore

Pizza made with ingredients produced by concrete factory.



More pizza for Ethan and Joel...

Ethan and Joel can't get enough Objectville Pizza! What they don't know is that now their orders are making use of the new ingredient factories. So now when they order...



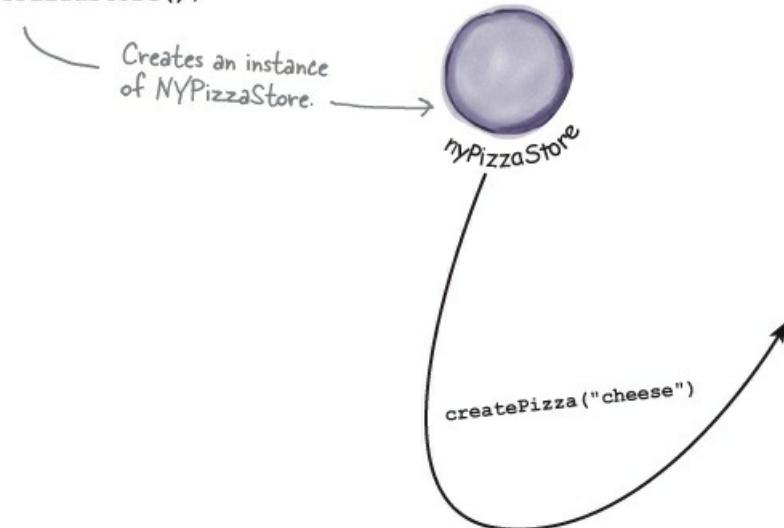
Behind the Scenes



The first part of the order process hasn't changed at all. Let's follow Ethan's order again:

- ① First we need a NY PizzaStore:

```
PizzaStore nyPizzaStore = new 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.

③ The orderPizza() method first calls the createPizza() method:

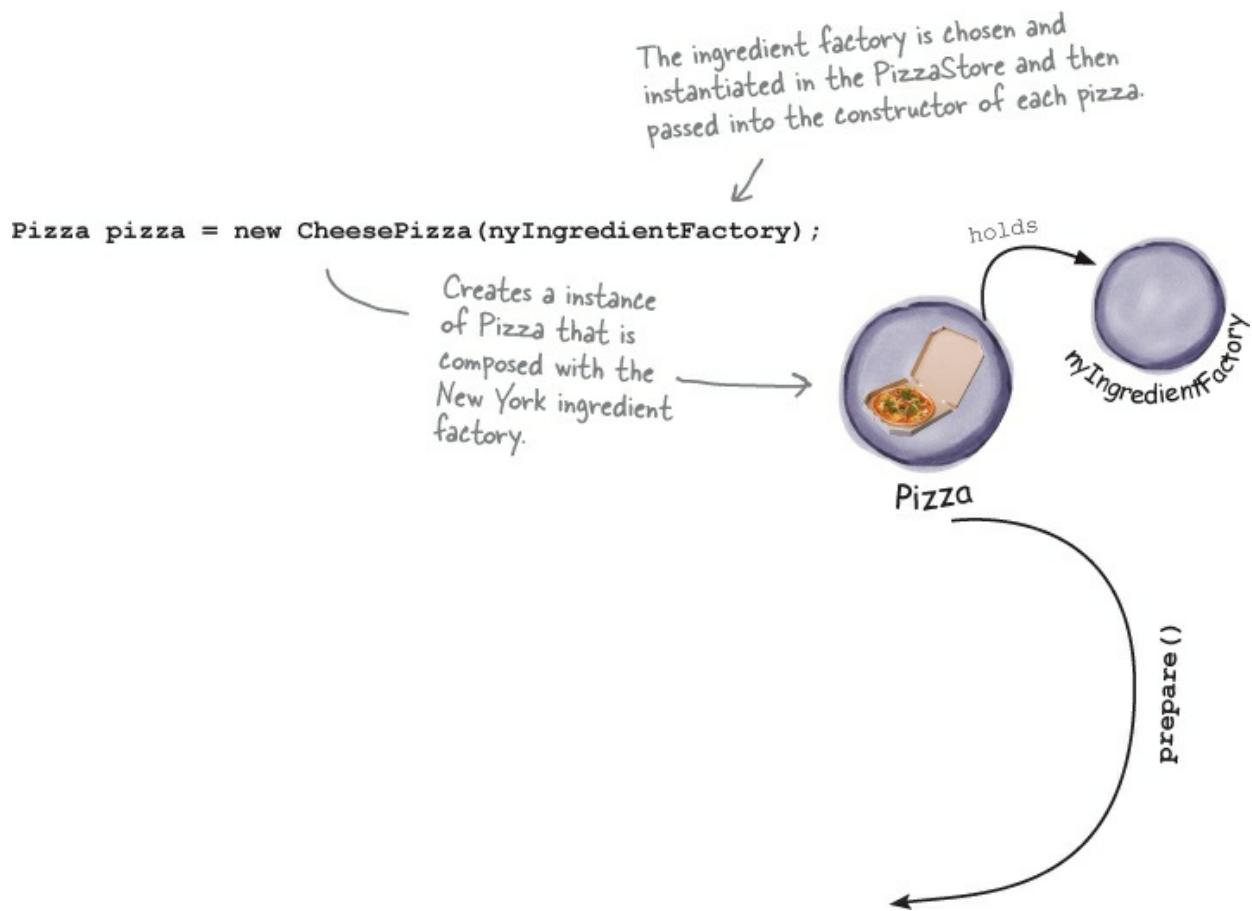
```
Pizza pizza = createPizza("cheese");
```

From here things change, because we are using an ingredient factory



Behind the Scenes

④ When the createPizza() method is called, that's when our ingredient factory gets involved:



⑤ Next we need to prepare the pizza. Once the `prepare()` method is called, the factory is asked to prepare ingredients:

```
void prepare() {
    dough = factory.createDough();
    sauce = factory.createSauce();
    cheese = factory.createCheese();
}
```

Thin crust

Marinara

Reggiano

For Ethan's pizza the New York ingredient factory is used, and so we get the NY ingredients.

⑥ Finally, we have the prepared pizza in hand and the `orderPizza()` method bakes, cuts, and boxes the pizza.

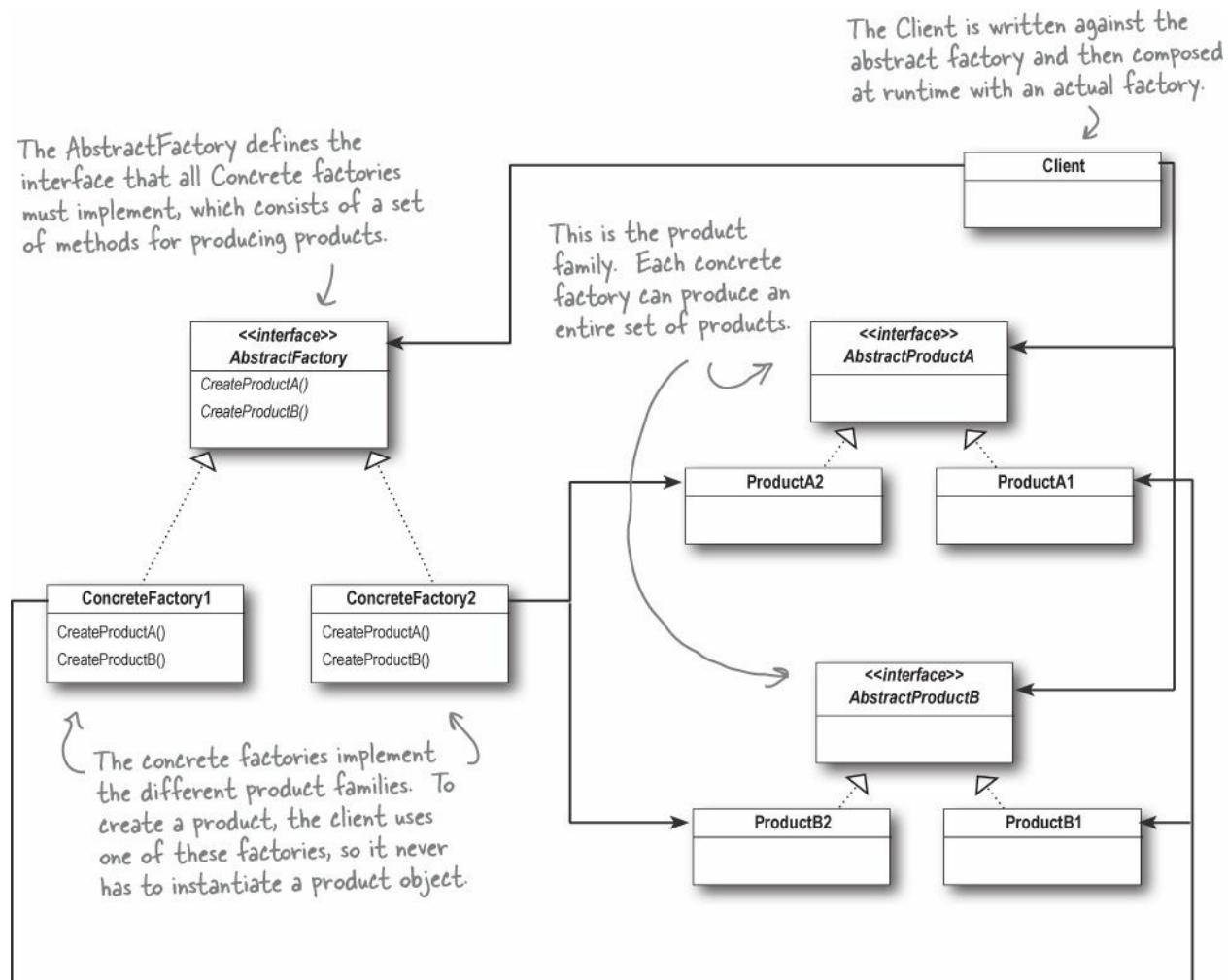
Abstract Factory Pattern defined

We're adding yet another factory pattern to our pattern family, one that lets us create families of products. Let's check out the official definition for this pattern:

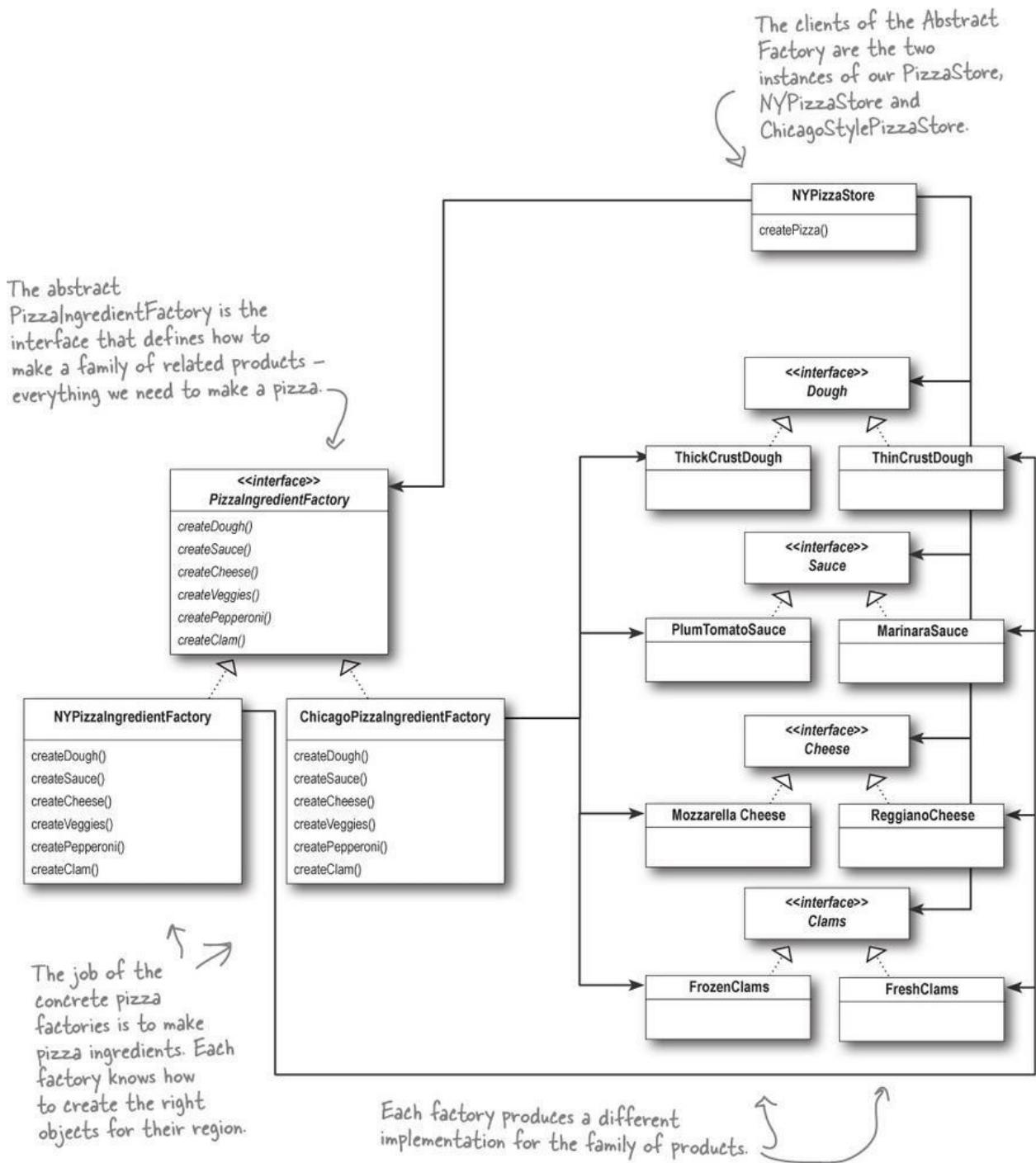
NOTE

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. Let's look at the class diagram to see how this all holds together:



That's a fairly complicated class diagram; let's look at it all in terms of our PizzaStore:





I noticed that each method in the Abstract Factory actually looks like a Factory Method (`createDough()`, `createSauce()`, etc.). Each method is declared abstract and the subclasses override it to create some object. Isn't that Factory Method?

Is that a Factory Method lurking inside the Abstract Factory?

Good catch! Yes, often the methods of an Abstract Factory are implemented as factory methods. It makes sense, right? 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.

PATTERNS EXPOSED

This week's interview: Factory Method and Abstract Factory, on each other

HeadFirst: Wow, an interview with two patterns at once! This is a first for us.

Factory Method: Yeah, I'm not so sure I like being lumped in with Abstract Factory, you know. Just because we're both factory patterns doesn't mean we shouldn't get our own interviews.

HeadFirst: Don't be miffed, we wanted to interview you together so we could help clear up any confusion about who's who for the readers. You do have similarities, and I've heard that people sometimes get you confused.

Abstract Factory: It is true, there have been times I've been mistaken for Factory Method, and I know you've had similar issues, Factory Method. We're both really good at decoupling applications from specific implementations; we just do it in different ways. So I can see why people might sometimes get us confused.

Factory Method: Well, it still ticks me off. After all, I use classes to create and you use objects; that's totally different!

HeadFirst: Can you explain more about that, Factory Method?

Factory Method: Sure. Both Abstract Factory and I create objects — that's our jobs. But I do it through inheritance...

Abstract Factory: ...and I do it through object composition.

Factory Method: Right. So that means, to create objects using Factory Method, you need to extend a class and provide an implementation for a factory method.

HeadFirst: And that factory method does what?

Factory Method: It creates objects, of course! I mean, the whole point of the Factory Method Pattern is that you're using a subclass to do your creation for you. In that way, clients only need to know the abstract type they are using, the subclass worries about the concrete type. So, in other words, I keep clients decoupled from the concrete types.

Abstract Factory: And I do too, only I do it in a different way.

HeadFirst: Go on, Abstract Factory... you said something about object composition?

Abstract Factory: 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. So, like Factory Method, my clients are decoupled from the actual concrete products they use.

HeadFirst: Oh, I see, so another advantage is that you group together a set of related products.

Abstract Factory: That's right.

HeadFirst: What happens if you need to extend that set of related products to, say, add another one? Doesn't that require changing your interface?

Abstract Factory: That's true; my interface has to change if new products are added, which I know people don't like to do....

Factory Method: <snicker>

Abstract Factory: What are you snickering at, Factory Method?

Factory Method: Oh, come on, that's a big deal! Changing your interface means you have to go in and change the interface of every subclass! That sounds like a lot of work.

Abstract Factory: Yeah, but I need a big interface because I am used to creating entire families of products. You're only creating one product, so you don't really need a big interface, you just need one method.

HeadFirst: Abstract Factory, I heard that you often use factory methods to implement your concrete factories?

Abstract Factory: Yes, I'll admit it, my concrete factories often implement a factory method to create their products. In my case, they are used purely to create products...

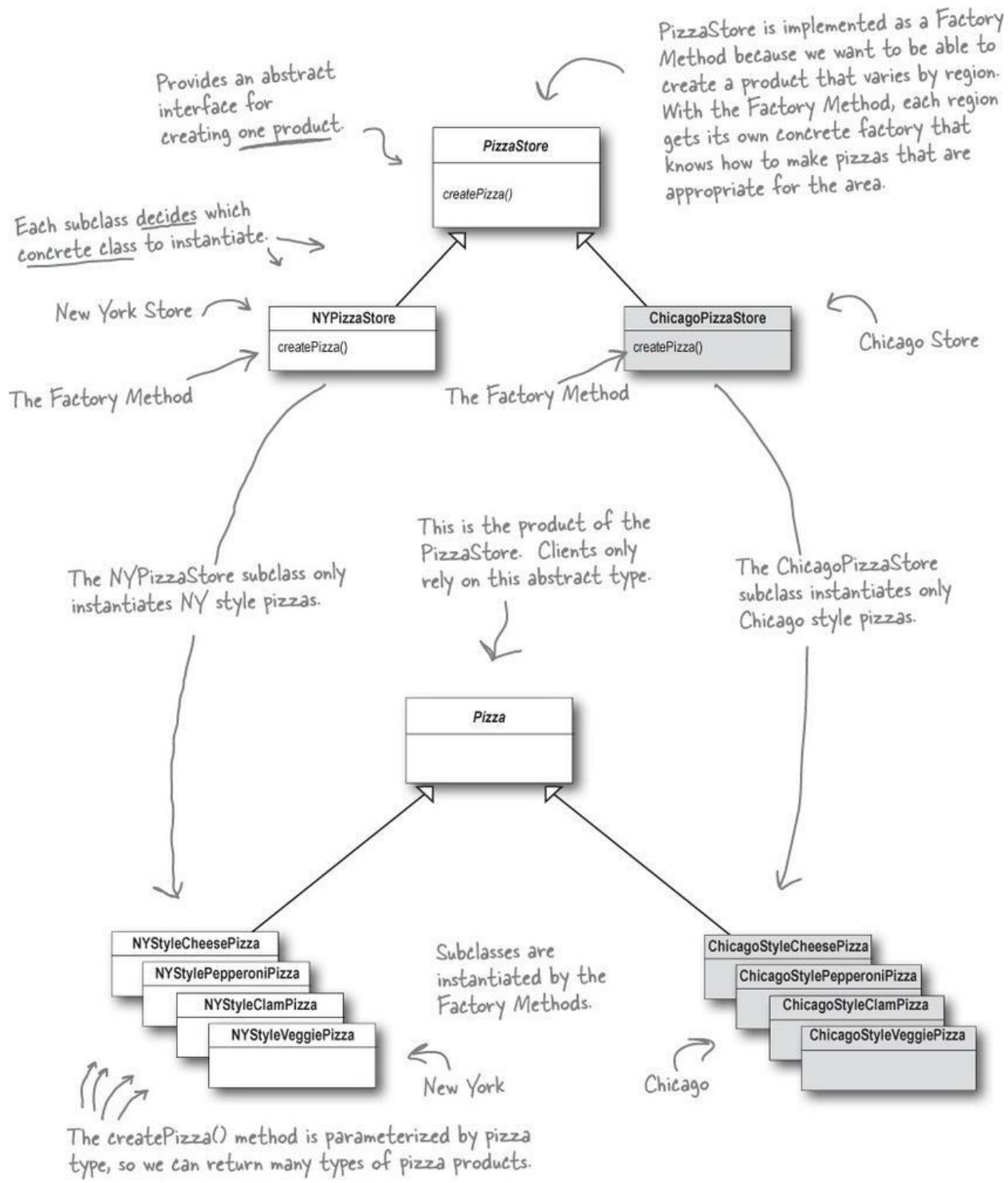
Factory Method: ...while in my case I usually implement code in the abstract creator that makes use of the concrete types the subclasses create.

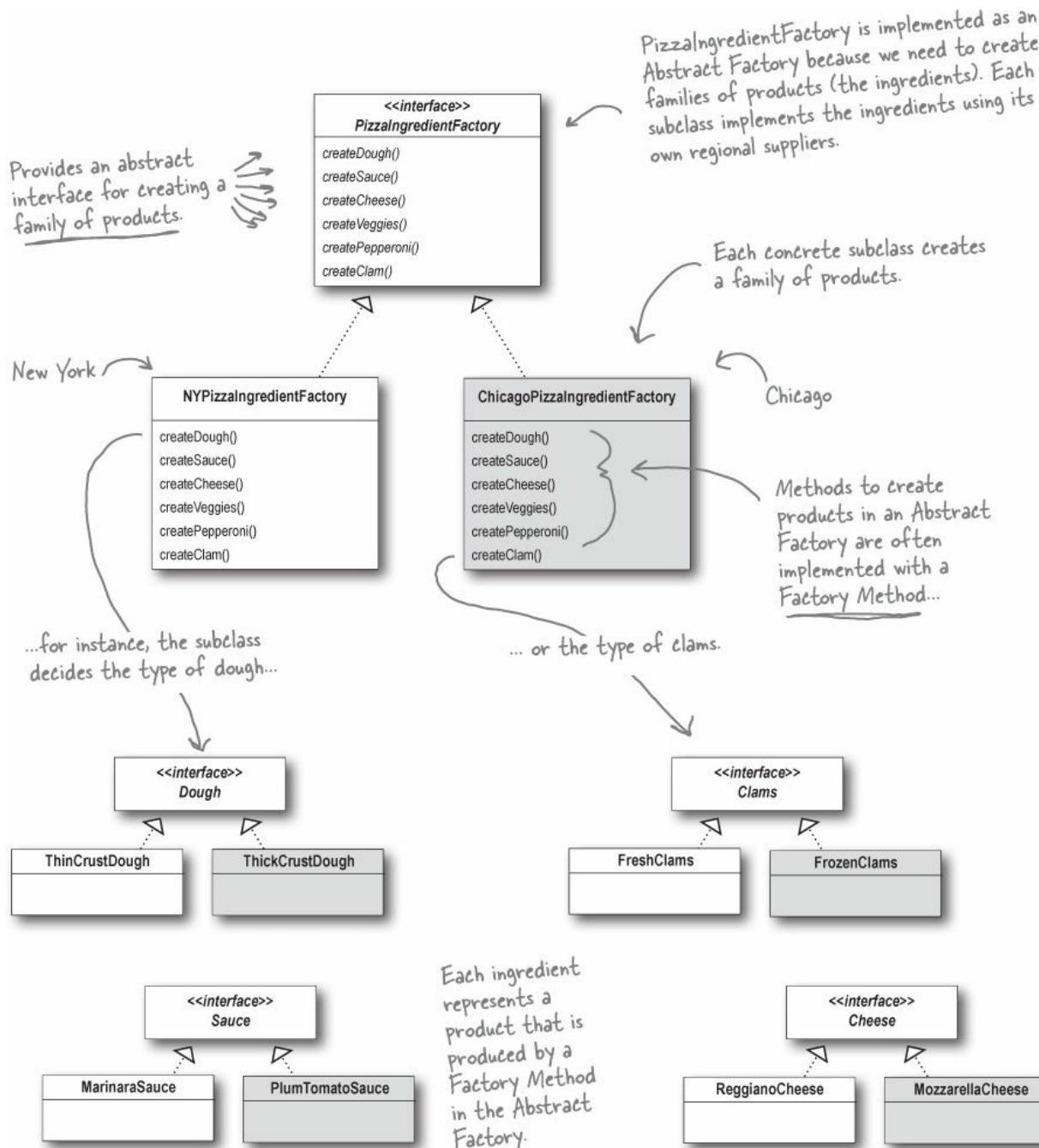
HeadFirst: It sounds like you both are good at what you do. I'm sure people like having a choice; after all, factories are so useful, they'll want to use them in all kinds of different situations. You both encapsulate object creation to keep applications loosely coupled and less dependent on implementations, which is really great, whether you're using Factory Method or Abstract Factory. May I allow you each a parting word?

Abstract Factory: Thanks. Remember me, Abstract Factory, and 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.

Factory Method: And I'm Factory Method; 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!

Factory Method and Abstract Factory compared





NOTE

The product subclasses create parallel sets of product families. Here we have a New York ingredient family and a Chicago family.

Tools for your Design Toolbox

In this chapter, we added two more tools to your toolbox: Factory Method and Abstract Factory. Both patterns encapsulate object creation and allow you to decouple your code from concrete types.

OO Basics

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

Depend on abstractions. Do not depend on concrete classes.

straction
apsulation
ymorphism
eritance

We have a new principle that guides us to keep things abstract whenever possible.

OO Patterns

S
c
I
in
va
z
a
f

Abstract Factory - Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Factory Method - Defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to the subclasses.

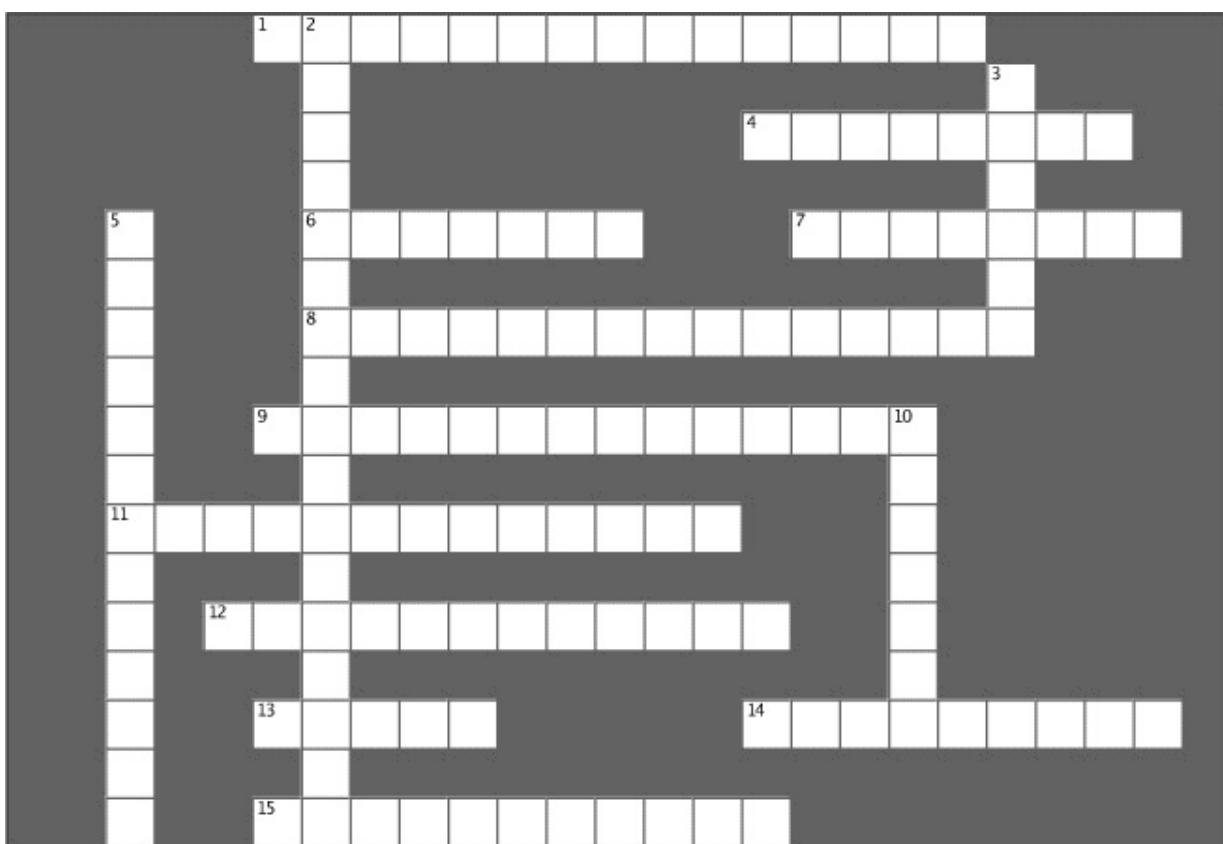
Both of these new patterns encapsulate object creation and lead to more decoupled, flexible designs.

BULLET POINTS

- All factories encapsulate object creation.
- Simple Factory, while not a bona fide design pattern, is a simple way to decouple your clients from concrete classes.
- Factory Method relies on inheritance: object creation is delegated to subclasses, which implement the factory method to create objects.
- Abstract Factory relies on object composition: object creation is implemented in methods exposed in the factory interface.
- All factory patterns promote loose coupling by reducing the dependency of your application on concrete classes.
- The intent of Factory Method is to allow a class to defer instantiation to its subclasses.
- The intent of Abstract Factory is to create families of related objects without having to depend on their concrete classes.
- The Dependency Inversion Principle guides us to avoid dependencies on concrete types and to strive for abstractions.
- Factories are a powerful technique for coding to abstractions, not concrete classes.

DESIGN PATTERNS CROSSWORD

It's been a long chapter. Grab a slice of Pizza and relax while doing this crossword; all of the solution words are from this chapter.



Across	Down
1. In Factory Method, each franchise is a _____.	2. We used _____ in Simple Factory and Abstract Factory, and inheritance in Factory Method.
4. In Factory Method, who decides which class to instantiate?	3. Abstract Factory creates a _____ of products.
6. Role of PizzaStore in Factory Method Pattern.	5. Not a REAL factory pattern, but handy nonetheless.
7. All New York style pizzas use this kind of cheese.	10. Ethan likes this kind of pizza.
8. In Abstract Factory, each ingredient factory is a _____.	
9. When you use new, you are programming to an _____.	
11. createPizza() is a _____ (two words).	
12. Joel likes this kind of pizza.	
13. In Factory Method, the PizzaStore and the concrete Pizzas all depend on this abstraction.	
14. When a class instantiates an object from a	

concrete class, it's _____ on that object.

15. All factory patterns allow us to _____ object creation.

SHARPEN YOUR PENCIL SOLUTION

We've knocked out the NY Pizza Store; just two more to go and we'll be ready to franchise! Write the Chicago and California Pizza Store implementations here:

Both of these stores are almost exactly like the New York store... they just create different kinds of pizzas.

```
public class ChicagoPizzaStore extends PizzaStore {  
    protected Pizza createPizza(String item) {  
        if (item.equals("cheese")) {  
            return new ChicagoStyleCheesePizza();  
        } else if (item.equals("veggie")) {  
            return new ChicagoStyleVeggiePizza();  
        } else if (item.equals("clam")) {  
            return new ChicagoStyleClamPizza();  
        } else if (item.equals("pepperoni")) {  
            return new ChicagoStylePepperoniPizza();  
        } else return null;  
    }  
}
```

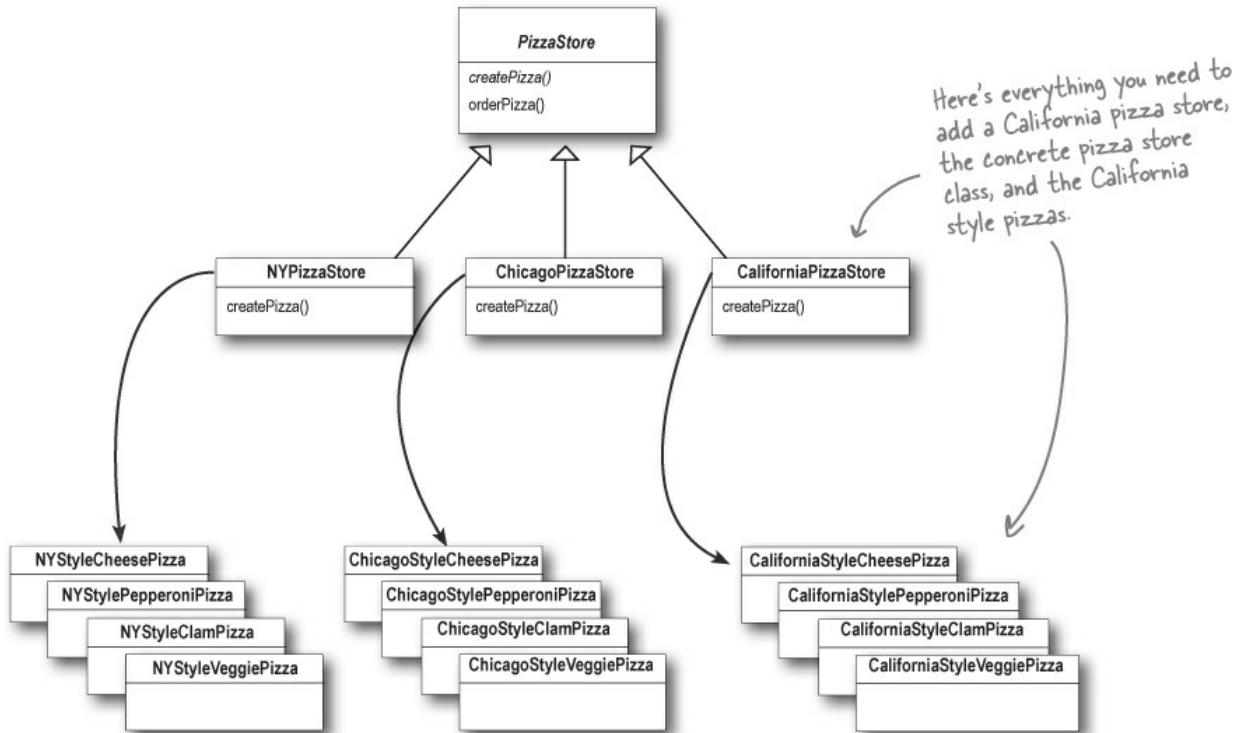
For the Chicago pizza store, we just have to make sure we create Chicago style pizzas...

```
public class CaliforniaPizzaStore extends PizzaStore {  
    protected Pizza createPizza(String item) {  
        if (item.equals("cheese")) {  
            return new CaliforniaStyleCheesePizza();  
        } else if (item.equals("veggie")) {  
            return new CaliforniaStyleVeggiePizza();  
        } else if (item.equals("clam")) {  
            return new CaliforniaStyleClamPizza();  
        } else if (item.equals("pepperoni")) {  
            return new CaliforniaStylePepperoniPizza();  
        } else return null;  
    }  
}
```

and for the California pizza store, we create California style pizzas.

DESIGN PUZZLE SOLUTION

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.



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

NOTE

Here are our suggestions...

Mashed Potatoes with Roasted Garlic

BBQ Sauce

Artichoke Hearts

M&M's

Peanuts

A very dependent PizzaStore

SHARPEN YOUR PENCIL SOLUTION

Let's pretend you've never heard of an OO factory. Here's a version of the PizzaStore that doesn't use a factory; make a count of the number of concrete pizza objects this class is dependent on. If you added California style pizzas to this PizzaStore, how many objects would it be dependent on then?

```
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

You can write your answers here:

8 number

12 number with California too

SHARPEN YOUR PENCIL SOLUTION

Go ahead and write the ChicagoPizzaIngredientFactory; you can reference the classes below in your implementation:

```
public class ChicagoPizzaIngredientFactory
```

```

    implements PizzaIngredientFactory
{

    public Dough createDough() {
        return new ThickCrustDough();
    }

    public Sauce createSauce() {
        return new PlumTomatoSauce();
    }

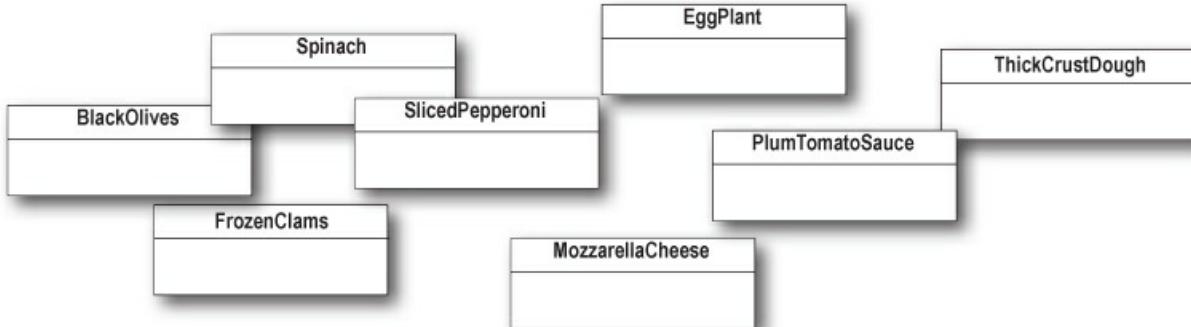
    public Cheese createCheese() {
        return new MozzarellaCheese();
    }

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new BlackOlives(),
                             new Spinach(),
                             new Eggplant() };
        return veggies;
    }

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

    public Clams createClam() {
        return new FrozenClams();
    }
}

```



DESIGN PATTERNS CROSSWORD SOLUTION

It's been a long chapter. Grab a slice of Pizza and relax while doing this crossword; all of the solution words are from this chapter. Here's the solution.

