there are no Dumb Questions

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: 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: 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: 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, we typically create decorators 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: 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: 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 decription 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();
}
```