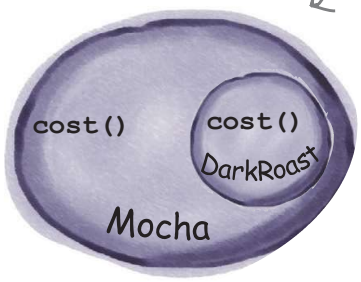
Constructing a drink order with Decorators

1 We start with our DarkRoast object.

Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

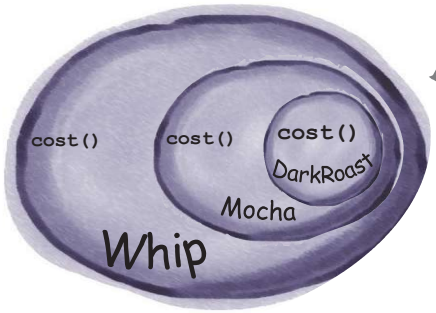
2 The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.

The Mocha object is a decorator. Its type mirrors the object it is decorating—in this case, a Beverage. (By "mirror," we mean it is the same type.)



So, Mocha has a cost() method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

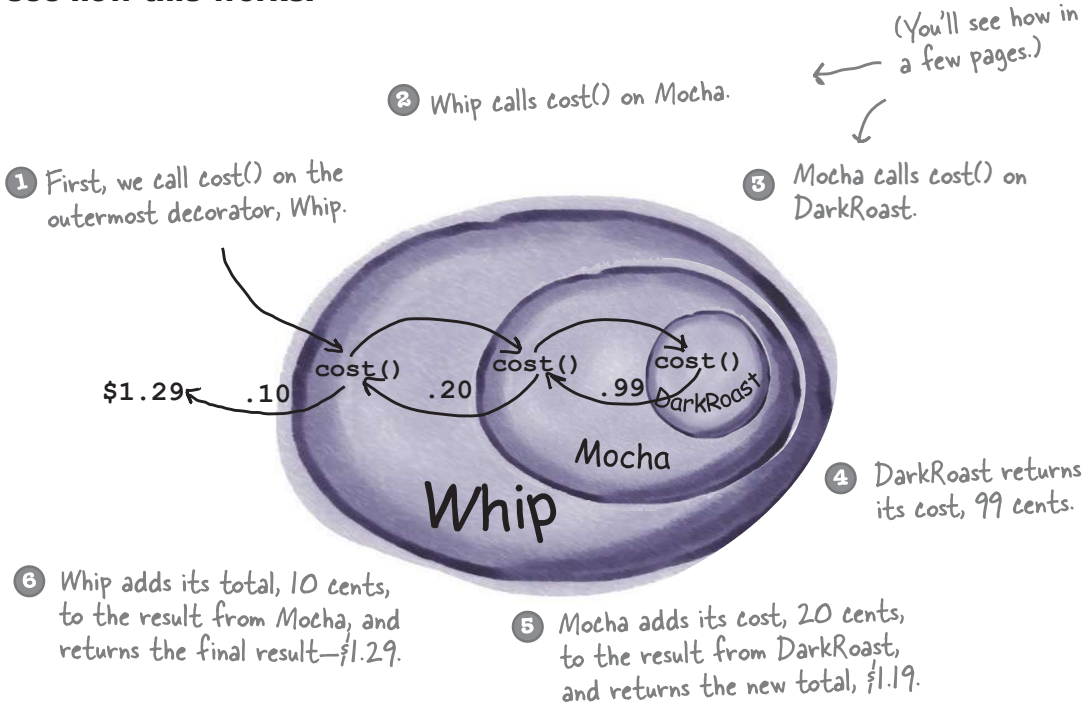
3 The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.



Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

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. And so on. Let's see how this works:



Okay, here's what we know about Decorators, 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 before and/or after delegating to the object it decorates to do the rest of the job.

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.