Now, I would have thought that functionality like asking the customer could have been used by all subclasses?

You know what? We agree with you. But you have to admit before you thought of that, it was a pretty cool example of how a hook can be used to conditionally control the flow of the algorithm in the abstract class. Right?

We're sure you can think of many other more realistic scenarios where you could use the template method and hooks in your own code.

there are no
Dumb Questions

Q: **When I'm creating a template method, how do I know when to use abstract methods and when to use hooks?**

A: Use abstract methods when your subclass MUST provide an implementation of the method or step in the algorithm. Use hooks when that part of the algorithm is optional. With hooks, a subclass may choose to implement that hook, but it doesn't have to.

Q: **What are hooks really supposed to be used for?**

A: There are a few uses of hooks. As we just said, a hook may provide a way for a subclass to implement an optional part of an algorithm, or if it isn't important to the subclass's implementation, it can skip it. Another use is to give the subclass a chance to react to some step in the template method that is about to happen or just happened. For instance, a hook method like justReorderedList() allows the subclass to perform some activity (such as redisplaying an onscreen representation) after an internal list is reordered. As you've seen, a hook can also provide a subclass with the ability to make a decision for the abstract class.

Q: **Does a subclass have to implement all the abstract methods in the AbstractClass?**

A: Yes, each concrete subclass defines the entire set of abstract methods and provides a complete implementation of the undefined steps of the template method's algorithm.

Q: **It seems like I should keep my abstract methods small in number; otherwise, it will be a big job to implement them in the subclass.**

A: That's a good thing to keep in mind when you write template methods. Sometimes you can do this by not making the steps of your algorithm too granular. But it's obviously a tradeoff: the less granularity, the less flexibility.

Remember, too, that some steps will be optional, so you can implement these as hooks rather than abstract methods, easing the burden on the subclasses of your abstract class.