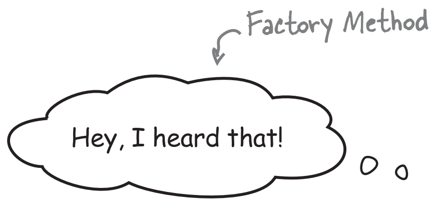
Tonight's talk: **Template Method and Strategy compare methods.**

Template Method:

Hey Strategy, what are you doing in my chapter? I figured I'd get stuck with someone boring like Factory Method.

Strategy:



Nope, it's me, although be careful—you and Factory Method are related, aren't you?

I was just kidding! But seriously, what are you doing here? We haven't heard from you in seven chapters!

I'd heard you were on the final draft of your chapter and I thought I'd swing by to see how it was going. We have a lot in common, so I thought I might be able to help...

You might want to remind the reader what you're all about, since it's been so long.

I don't know, since Chapter 1, people have been stopping me in the street saying, "Aren't you that pattern...?" So I think they know who I am. But for your sake: I define a family of algorithms and make them interchangeable. Since each algorithm is encapsulated, the client can use different algorithms easily.

Hey, that does sound a lot like what I do. But my intent's a little different from yours; my job is to define the outline of an algorithm, but let my subclasses do some of the work. That way, I can have different implementations of an algorithm's individual steps, but keep control over the algorithm's structure. Seems like you have to give up control of your algorithms.

I'm not sure I'd put it quite like *that*...and anyway, I'm not stuck using inheritance for algorithm implementations. I offer clients a choice of algorithm implementation through object composition.

Template Method:

Strategy:

I remember that. But I have more control over my algorithm and I don't duplicate code. In fact, if every part of my algorithm is the same except for, say, one line, then my classes are much more efficient than yours. All my duplicated code gets put into the superclass, so all the subclasses can share it.

You might be a little more efficient (just a little) and require fewer objects. *And* you might also be a little less complicated in comparison to my delegation model, but I'm more flexible because I use object composition. With me, clients can change their algorithms at runtime simply by using a different strategy object. Come on, they didn't choose me for Chapter 1 for nothing!

Yeah, well, I'm *real* happy for ya, but don't forget I'm the most used pattern around. Why? Because I provide a fundamental method for code reuse that allows subclasses to specify behavior. I'm sure you can see that this is perfect for creating frameworks.

Yeah, I guess...but what about dependency? You're way more dependent than me.

How's that? My superclass is abstract.

But you have to depend on methods implemented in your subclasses, which are part of your algorithm. I don't depend on anyone; I can do the entire algorithm myself!

Like I said, Strategy, I'm *real* happy for you. Thanks for stopping by, but I've got to get the rest of this chapter done.

Okay, okay, don't get touchy. I'll let you work, but let me know if you need my special techniques anyway; I'm always glad to help.

Got it. Don't call us, we'll call you...