there are no Dumb Questions

Q: For such a simple pattern consisting of only one class, Singleton sure seems to have some problems.

A: Well, we warned you up front! But don't let the problems discourage you; while implementing Singletons correctly can be tricky, after reading this chapter you're now well informed on the techniques for creating Singletons and should use them wherever you need to control the number of instances you're creating.

Q: Can't I just create a class in which all methods and variables are defined as static? Wouldn't that be the same as a Singleton?

A: Yes, if your class is self-contained and doesn't depend on complex initialization. However, because of the way static initializations are handled in Java, this can get very messy, especially if multiple classes are involved. Often this scenario can result in subtle, hard-to-find bugs involving order of initialization. Unless there is a compelling need to implement your "singleton" this way, it's far better to stay in the object world.

Q: What about class loaders? I heard there's a chance that two class loaders could each end up with their own instance of Singleton.

A: Yes, that is true as each class loader defines a namespace. If you have two or more class loaders, you can load the same class multiple times (once in each class loader). Now, if that class happens to be a Singleton, then since we have more than one version of the class, we also have more than one instance of Singleton. So, if you are using multiple class loaders and Singletons, be careful. One way around this problem is to specify the class loader yourself.

Q: And reflection, and serialization/deserialization?

A: Yes, reflection and serialization/deserialization can also present problems with Singletons. If you're an advanced Java user using reflection, serialization, and deserialization, you'll need to keep that in mind.

Q: Earlier we talked about the loose coupling principle. Isn't a Singleton violating this? After all, every object in our code that depends on the Singleton is going to be tightly coupled to that very specific object.

A: Yes, and in fact this is a common criticism of the Singleton Pattern. The loose coupling principle says to "strive for loosely coupled designs between objects that interact." It's easy for Singletons to violate this principle: if you make a change to the Singleton, you'll likely have to make a change to every object connected to it.

Q: I've always been taught that a class should do one thing and one thing only. For a class to do two things is considered bad OO design. Isn't a Singleton violating this too?

A: You would be referring to the Single Responsibility Principle, and yes, you are correct: the Singleton is responsible not only for managing its one instance (and providing global access), but also for whatever its main role is in your application. So, certainly you could argue it is taking on two responsibilities. Nevertheless, it isn't hard to see that there is utility in a class managing its own instance; it certainly makes the overall design simpler. In addition, many developers are familiar with the Singleton Pattern as it is in wide use. That said, some developers do feel the need to abstract out the Singleton functionality.

Q: I wanted to subclass my Singleton code, but I ran into problems. Is it okay to subclass a Singleton?

A: One problem with subclassing a Singleton is that the constructor is private. You can't extend a class with a private constructor. So, the first thing you'll have to do is change your constructor so that it's public or protected. But then, it's not really a Singleton anymore, because other classes can instantiate it.

If you do change your constructor, there's another issue. The implementation of Singleton is based on a static variable, so if you do a straightforward subclass, all of your derived classes will share the same instance variable. This is probably not what you had in mind. So, for subclassing to work, implementing a registry of sorts is required in the base class.

But what are you really gaining from subclassing a Singleton? Like most patterns, Singleton is not necessarily meant to be a solution that can fit into a library. In addition, the Singleton code is trivial to add to any existing class. Last, if you are using a large number of Singletons in your application, you should take a hard look at your design. Singletons are meant to be used sparingly.

Q: I still don't totally understand why global variables are worse than a Singleton.

A: In Java, global variables are basically static references to objects. There are a couple of disadvantages to using global variables in this manner. We've already mentioned one: the issue of lazy versus eager instantiation. But we need to keep in mind the intent of the pattern: to ensure only one instance of a class exists and to provide global access. A global variable can provide the latter, but not the former. Global variables also tend to encourage developers to pollute the namespace with lots of global references to small objects. Singletons don't encourage this in the same way, but can be abused nonetheless.