**Comparing composition and inheritance**  
So how exactly do composition and inheritance compare? Here are several points of comparison:

* It is easier to change the interface of a back-end class (composition) than a superclass (inheritance). As the previous example illustrated, a change to the interface of a back-end class necessitates a change to the front-end class implementation, but not necessarily the front-end interface. Code that depends only on the front-end interface still works, so long as the front-end interface remains the same. By contrast, a change to a superclass's interface can not only ripple down the inheritance hierarchy to subclasses, but can also ripple out to code that uses just the subclass's interface.
* It is easier to change the interface of a front-end class (composition) than a subclass (inheritance). Just as superclasses can be fragile, subclasses can be rigid. You can't just change a subclass's interface without making sure the subclass's new interface is compatible with that of its supertypes. For example, you can't add to a subclass a method with the same signature but a different return type as a method inherited from a superclass. Composition, on the other hand, allows you to change the interface of a front-end class without affecting back-end classes.
* Composition allows you to delay the creation of back-end objects until (and unless) they are needed, as well as changing the back-end objects dynamically throughout the lifetime of the front-end object. With inheritance, you get the image of the superclass in your subclass object image as soon as the subclass is created, and it remains part of the subclass object throughout the lifetime of the subclass.
* It is easier to add new subclasses (inheritance) than it is to add new front-end classes (composition), because inheritance comes with polymorphism. If you have a bit of code that relies only on a superclass interface, that code can work with a new subclass without change. This is not true of composition, unless you use composition with interfaces. Used together, composition and interfaces make a very powerful design tool. I'll talk about this approach in next month's **Design Techniques** article.
* The explicit method-invocation forwarding (or delegation) approach of composition will often have a performance cost as compared to inheritance's single invocation of an inherited superclass method implementation. I say "often" here because the performance really depends on many factors, including how the JVM optimizes the program as it executes it.
* With both composition and inheritance, changing the implementation (not the interface) of any class is easy. The ripple effect of implementation changes remain inside the same class.

***Choosing between composition and inheritance  
So how do all these comparisons between composition and inheritance help you in your designs? Here are a few guidelines that reflect how I tend to select between composition and inheritance.***

* **Make sure inheritance models the *is-a* relationship**  
  My main guiding philosophy is that inheritance should be used only when a subclass *is-a* superclass. In the example above, an Apple likely is-a Fruit, so I would be inclined to use inheritance.
* An important question to ask yourself when you think you have an is-a relationship is whether that is-a relationship will be constant throughout the lifetime of the application and, with luck, the lifecycle of the code. For example, you might think that an Employee is-a Person, when really Employee represents a role that a Person plays part of the time. What if the person becomes unemployed? What if the person is both an Employee and a Supervisor? Such impermanent is-a relationships should usually be modelled with composition.
* **Don't use inheritance just to get code reuse**  
  If all you really want is to reuse code and there is no is-a relationship in sight, use composition.
* **Don't use inheritance just to get at polymorphism**  
  If all you really want is polymorphism, but there is no natural is-a relationship, use composition with interfaces. I'll be talking about this subject next month.

Composition vs. Inheritance

Inheritance Yields (Slightly) Better Performance:

* Composition's method forwarding/delegation will often be slower than inheritance's dynamic binding
* Composition results in more objects getting instantiated, which can incur a performance cost at allocation, <init>(), and GC time

Composition Yields Better Flexibility:

* Interfaces of classes involved in a composition relationship need not be compatible, so it's easier to change the interfaces
* Composition allows you to delay creation of back-end objects until (and unless) you need them
* Composition allows you to change back-end objects throughout the lifetime of the front-end object
* Composition allows front-end objects to share the same back-end objects

But:

* Composition's method forwarding/delegation results in more code that has to be written, debugged, and maintained.
* Easier to add new subclasses than new front-end classes, *unless you use composition with interfaces*