[instanceof Vs getClass( )](http://stackoverflow.com/questions/4989818/instanceof-vs-getclass)

<http://stackoverflow.com/questions/4989818/instanceof-vs-getclass>

The reason that the performance of instanceof and getClass() == ... is different is that they are doing different things.

* instanceof tests whether the object reference on the left-hand side (LHS) is an instance of the type on the right-hand side (RHS) or some subtype.
* getClass() == ... tests whether the types are identical.

So the recommendation is to ignore the performance issue and use the alternative that gives you the answer that you need.

And yes, overuse of either of them is "design smell". If you are not careful, you end up with a design where the addition of new subclasses results in a significant amount of code reworking. In most situations, the preferred approach is to use polymorphism.

<http://stackoverflow.com/questions/596462/any-reason-to-prefer-getclass-over-instanceof-when-generating-equals>

The reason that I favor the instanceof approach is that when you use the getClass approach, you have the restriction that objects are only equal to other objects of the same class, the same run time type. If you extend a class and add a couple of innocuous methods to it, then check to see whether some object of the subclass is equal to an object of the super class, even if the objects are equal in all important aspects, you will get the surprising answer that they aren't equal. In fact, this violates a strict interpretation of the Liskov substitution principle, and can lead to very surprising behavior. In Java, it's particularly important because most of the collections (HashTable, etc.) are based on the equals method. If you put a member of the super class in a hash table as the key and then look it up using a subclass instance, you won't find it, because they are not equal.

Angelika Langers [**Secrets of equals**](http://www.angelikalanger.com/Articles/JavaSolutions/SecretsOfEquals/Equals.html) gets into that with a long and detailed discussion for a few common and well-known examples, including by Josh Bloch and Barbara Liskov, discovering a couple of problems in most of them. She also gets into the instanceof vs getClass. Some quote from it

Conclusions

Having dissected the four arbitrarily chosen examples of implementations of equals() , what do we conclude?

First of all: there are two substantially different ways of performing the check for type match in an implementation of equals() . A class can allow mixed-type comparison between super- and subclass objects by means of the instanceof operator, or a class can treat objects of different type as non-equal by means of the getClass() test. The examples above illustrated nicely that implementations of equals() using getClass() are generally more robust than those implementations using instanceof .

The instanceof test is correct only for final classes or if at least method equals() is final in a superclass. The latter essentially implies that no subclass must extend the superclass's state, but can only add functionality or fields that are irrelevant for the object's state and behavior, such as transient or static fields.

Implementations using the getClass() test on the other hand always comply to the equals() contract; they are correct and robust. They are, however, semantically very different from implementations that use the instanceof test. Implementations using getClass() do not allow comparison of sub- with superclass objects, not even when the subclass does not add any fields and would not even want to override equals() . Such a "trivial" class extension would for instance be the addition of a debug-print method in a subclass defined for exactly this "trivial" purpose. If the superclass prohibits mixed-type comparison via the getClass() check, then the trivial extension would not be comparable to its superclass. Whether or not this is a problem fully depends on the semantics of the class and the purpose of the extension.