How Do I Correctly Implement the *equals()* Method?

Tal Cohen

The Java *equals()* method, which is defined in *java.lang.Object*, is used for instance equality testing (as opposed to reference equality, which is tested using the == operator). Consider, for example, these two assignments:

```
Date d1 = new Date(2001, 10, 27);
Date d2 = new Date(2001, 10, 27);
```

In this case, *d1 == d2* returns False (since == tests for reference equality, and the two variables are references to different objects). However, *d1.equals(d2)* returns True.

The default implementation of *equals()* is based on the == operator: Two objects are equal if and only if they are the same object. Naturally, most classes should define their own alternative implementation of this important method.

However, implementing *equals()* correctly is not straightforward. The *equals()* method has a contract that says the equality relation must meet these demands:

- It must be reflexive. For any reference *x*, *x.equals(x)* must return True.
- It must be symmetric. For any two nonnull references *x* and *y*, *x.equals(y)* should return the exact same value as *y.equals(x)*.
- It must be transitive. For any three references *x*, *y*, and *z*, if *x.equals(y)* and *y.equals(z)* are True, then *x.equals(z)* must also return True.
- It should be consistent. For any two references *x* and *y*, *x.equals(y)* should return the same value if called repeatedly (unless, of course, either *x* or *y* were changed between the repeated invocations of *equals()*).
- For any nonnull reference *x*, *x.equals-(null)* should return False.

This doesn't sound complicated: The first three items are the natural mathematical properties of equality, and the last two are trivial programmatic requirements. It looks like any implementation based on simple field-by-field comparison would do the trick. For example, in Listing One the class *Point* represents a point in two-dimensional space, with a suggested implementation for the *equals()* method. At first glance, it looks as though Listing One meets all five demands placed by the contract:

- It is reflexive, since whenever the parameter *o* is actually *this* (which is what happens when one invokes it using *x.equals(x)*), the fields match and the result is True.
- It seems symmetric. If some *Point* object *p1* finds its fields are equal to those of some other *Point* object *p2*, then *p2* would also find that its own fields are equal to those of *p1*. For example, after the two assignments:

```
Point p1 = new Point(1, 2);
Point p2 = new Point(1, 2);
```

both *p1.equals(p2)* and *p2.equals(p1)* return True. If, on the other hand, *p2* is different than *p1*, both calls return False.

- It seems transitive, for the same reasons.
- It is clearly consistent.
- Any call of the form *x.equals(null)* returns False, thanks to the test at the beginning of the code: If the parameter is not an instance of the class *Point*, the method returns False immediately. Since, in particular, null is not an instance of *Point* (nor indeed of any other class), the condition is met.

However, this is a naïve implementation. As Joshua Bloch shows in his book *Effective Java Programming Language Guide* (Addison-Wesley, 2001), things get much more complex when inheritance is involved.

Bloch presents the class *ColorPoint* (Listing Two), which extends *Point* and adds an aspect (namely, a new field). If *ColorPoint* implements *equals()* similarly to its superclass *Point*, symmetry is violated. Again, the implementation seems straightforward and correct. The problem arises when two objects are involved, each of a different class:

```
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);
```

Now, *p2.equals(p1)* returns True, since the two fields *p2*'s *equals()* method compares, *x* and *y*, are indeed equal. Yet *p1.equals(p2)* returns False because *p2* is not an instance of the *ColorPoint* class.

It is important to understand that an incorrect implementation of *equals()*, like that just presented, would cause problems in many unexpected places; for example, when the objects are used in various collection classes (that is, in their containment tests). And you have just seen that this simple implementation does not provide symmetry.

Listing Three, an alternative implementation of *equals()*, does meet the symmetry requirement. While at first it might seem a better solution, Bloch shows that it is broken, too. Symmetry is indeed preserved. *p1* and *p2* (from the earlier example) would both provide the same answer when asked if one equals the other. However, this implementation violates the demand for "transitivity." To see how, add a third reference, *p3*:

```
ColorPoint p3 = new ColorPoint(1, 2, Col or.BLUE);
```

In this case, *p1.equals(p2)* returns True, since *p1* realizes *p2* is not a *ColorPoint* and performs a color-blind comparison. *p2.equals(p3)* also returns True, since *p2*, being a simple *Point*, compares only the *x* and *y* fields and finds them to be equal. Transitivity demands that if *a*=*b* and *b*=*c*, then *a*=*c* as well. But in this case, even though *p1.equals(p2)* and *p2.equals(p3)*, the call *p1.equals(p3)* returns False.

Tal is a researcher in IBM's Haifa Research Labs in Israel. He can be contacted at tal@forum2.org.

One way to avoid the problem is to ignore any fields added in subclasses. This way, *ColorPoint* inherits the implementation of *equals()* provided by *Point*, and doesn't override it. This solution does meet all the contract demands for *equals()*. However, it is hardly a useful equality test; for example, *p1.equals(p3)* returns True, even though each point has a different color.

Bloch claims that "It turns out that this is a fundamental problem of equivalence relations in object-oriented languages. There is simply no way to extend an instantiable class and add an aspect while preserving the equals contract." He suggests that programmers use composition rather than inheritance to work around this problem. Taking this approach, the *ColorPoint* class would not extend *Point*, but rather include a field of that type, like Listing Four.

Is this the only solution? Not really. The *Point* class can be extended, adding an aspect, while preserving the *equals()* contract. The basic idea is this: For two objects to be equal, both must agree that they are equal. To prevent endless recursion during the mutual verification, you define a protected helper method, *blindlyEquals()*, which compares fields blindly. The *equals()* method then verifies that both objects agree that they are blindly equal to each other; see Listing Five. Note how the implementation of *blindlyEquals()* is simply the original implementation of *equals()*. However, *blindlyEquals()* is not bound by the *equals()* contract. By itself, it does not provide a symmetric comparison, but it does provide *equals()* with the services it needs to fully meet the contract demands.

In subclasses, you override *blindlyEquals()* only, leaving *equals()* unchanged. Listing Six, therefore, is a proper implementation of the class *ColorPoint*. Again, the implementation of *blindlyEquals()* is the original, nonsymmetric attempt to implement *equals()*. The *equals()* method itself is inherited from *Point*, and not overridden.

It is easy to see that this new implementation is both symmetric and transitive, as well as meeting all other demands placed by the *equals()* contract. In particular, when using the three objects defined in the previous examples:

- *p2.blindlyEquals(p1)* returns True, but *p1.blindlyEquals(p2)* returns False. Since *equals()* checks both ways, both *p1.equals(p2)* and *p2.equals(p1)* return False.
- Since *p1.equals(p2)* returns False (and *p2.equals(p3)* returns False as well), the transitivity demand does not hold in this case ($a \neq b$ and $b \neq c$ means you do not know in advance if $a=c$ or not).

It can be mathematically proven that symmetry and transitivity always hold with this implementation. The symmetry part is easy: For any two references *x* and *y*, *x.equals(y)* and *y.equals(x)* execute the same code (calling both *x.blindlyEquals(y)* and *y.blindlyEquals(x)*, although in a different order). Transitivity can be proven using *reductio ad absurdum*. And of course, the other three contract demands—reflexivity, consistency, and returning False when tested on null—are also met.

The technique presented here can be applied to any object hierarchy you define in Java. That *equals()* itself is never overridden means it would have been best if this implementation was part of the standard *java.lang.Object()* class, along with a default implementation of *blindlyEquals()*, which could be easily overridden by each subclass. However, since this change in the Java standard libraries is not likely to occur anytime soon, we will have to be content with manually including it in programs.

In short, whenever you define a new class, a definition of *blindlyEquals()* must be included as a nonsymmetric comparison operation, and an implementation of *equals()* (as presented here) should be added. Then, all subclasses of this newly defined class need only override *blindlyEquals()* to provide a complete, contract-abiding *equals()* comparison.

The method presented here can be used in any object-oriented language, and does not rely on run-time type information (other than the *instanceof* operator, which

The faster way to develop Windows Device Drivers.

BSQUARE's development toolkits save you valuable time and money by helping you create high-performance device drivers for Windows® XP/2000/98/Me/NT/NT Embedded.

WinDK™ toolkit: Develop high quality kernel-level device drivers with support for plug-and-play, power management, IEEE 1394, and USB for Windows operating systems.

WinRT™ toolkit: Create hardware control programs–even for USB devices–without using the DDK.

bSQUARE

Download a free evaluaton copy of BSQUARE's driver toolkits at www.bsquare.com/offer/download

©2001 BSQUARE Corporation. All rights reserved. BSQUARE is a registered trademark of BSQUARE Corporation. All other names, product names and trade names are trademarks or registered trademarks of their respective holders.