Equality versus identity

Bobby Woolf

ONE OF THE trickier concepts in Smalltalk is the distinction between object equality and object identity. Any experienced Smalltalk programmer knows the difference: If two objects are equal, "=" (equal) returns true; if they're identical, "==" (double-equal) returns true as well. But what does that difference actually mean in a logical programming sense? Should you design your code so that objects that are equal have to be identical? When should your code test with equal versus double-equal? The difference between equality and identity seems to be like fine art: You can't say what it is exactly but you know it when you see it. Even though you know what it is, it's very difficult to explain to someone else who doesn't know. Yet understanding this difference is important if you want to use Smalltalk well.

What exactly makes two objects equal is often very subjective.

A METAPHORICAL EXAMPLE

This little story illustrates the difference. (I can't take credit for this story; it's been floating around my company for years.) A person sits down at a restaurant table and notices the customer at the next table eating a delicious plate of lasagna. When the waiter arrives, the person points at the lasagna on the next table and says, "I want that." How should the waiter fulfill this request? He could get another plate of lasagna just like it and serve it. Or he could take the plate of lasagna away from the customer at the next table and serve it to this person.

If the waiter brought a second serving, the second plate of lasagna would be equivalent to the first one. If the waiter took the serving off the next table, the second plate of lasagna would be identical to the first one. In this context, equality means that the two plates have the same properties but are not the same plate. Identity means that the two plates are really the same plate. The most practical difference is that two equal plates contain two servings of lasagna; two identical plates contain a total of one serving.

SIGNIFICANCE IN SMALLTALK

When two seemingly separate objects have the same identity, that means they're really just two different handles on the same object. A *handle* is a variable or literal you use to access an object. The variable does not really contain the object, it just contains a pointer to the object. It is often said that everything in Smalltalk is an object. It could also be said that everything in Smalltalk is a pointer to an object. You can never actually work with the objects directly, you just work with the pointers to objects. Because everything is a pointer, there is never a need to distinguish between a pointer and the value itself.

POINTERS ARE SIMPLER

This explains some of the conventions that make Smalltalk code different (and simpler) than other languages. In C/C++, for example, a variable yourPhone might contain a PhoneNumber. If you copy yourPhone into myPhone (myPhone = yourPhone), whether the variables copy the value or share the same one depends on how the variables are declared, as shown in Listing 1.

C/C++ allows/forces you to manipulate the pointers specifically. Thus a programmer has to be careful to use them properly; a mistake can cause serious, often subtle bugs.

(By the way, you would not believe how much trouble I had writing that C++ example. I spoke to a couple of friends of mine who know C++ and told them the dilemma

Listing 1.

```
PhoneNumber myPhone, yourPhone;      // declares two instances of
                                        PhoneNumber
myPhone = yourPhone;                 // copies the PhoneNumber

PhoneNumber *myPhone, *yourPhone     // declares two pointers,
                                     // each points to a PhoneNumber

yourPhone = new PhoneNumber;         // initialization
myPhone = yourPhone;                 // shares the PhoneNumber
*myPhone = *yourPhone;               // copies the PhoneNumber
```

I was trying to illustrate. Our conversation then got very complicated as they explained all the different things that the programmer would have to consider. Just trying to find a simple example that shows two variables that share a single object versus copies of the object was very difficult. Somehow I think that the difficulty I had preparing this example does more than the example itself to illustrate how complex C/C++ can be!)

In Pascal, the programmer can declare a subroutine parameter to receive a variable either by value or by reference. If the variable is passed by value, it is copied, which is expensive for variables that contain a large amount of data. However, since the subroutine has its own local copy of the data, the subroutine can make all kinds of changes to the copy. It knows that those changes won't affect the rest of the program outside the subroutine. If a variable is passed by reference, the subroutine gets a pointer to the same data that the rest of the program is using. This pointer is inexpensive to create. However, the subroutine has to be more careful about the changes it makes to the parameter since those changes affect the original variable's data. Thus programmers that nonchalantly change a parameter's contents have to learn to be a lot more careful whenever that parameter is passed by reference.

Smalltalk programmers do not need to make these distinctions between a value and a pointer to the value. Any variable is just a pointer (see Listing 2).

So any Smalltalk variable declaration is equivalent to a C/C++ pointer variable declaration. Whereas Pascal parameters can be passed by reference (pointer) or value (copy), Smalltalk method parameters are always passed by reference. If you want to copy a variable's value in Smalltalk, you need to explicitly send it a message like copy. This makes Smalltalk's syntax simpler, its intent more explicit, and its code more consistent.

Listing 2.

```
| myPhone yourPhone |
yourPhone := PhoneNumber
    readFromString: '212-555-1010'.   "initialization"
myPhone := yourPhone.                 "shares the PhoneNumber"
myPhone := yourPhone copy.            "copies the PhoneNumber"
```

GARBAGE COLLECTION

Everything being a pointer helps explain how garbage collection works: When nobody's pointing to an object anymore, it gets garbage collected. Every variable is a pointer to a value. Code uses a variable as a handle to that object. As long as an object has a pointer to it, some code has a handle to access it, so it should not be garbage collected. But once there are no more pointers to an object, no one has a handle on it, so no one could access that object even if they wanted to. Since no one can access the object, it can safely be garbage collected and no one will miss it.

DISTINGUISHING OBJECTS

So how do you tell when two objects are actually the same object? You use the messages "=" (equal) and "==" (equal-equal or double-equal). *Equal* tells you whether two objects are equivalent, meaning that they represent two values where one is as good as the other. *Double-equal* tells you whether two objects are identical, which is to say that they are the same object. Equal tests for equality whereas double-equal tests for identity.

Double-equal is easy to design and implement. The double-equal method is defined in Object and cannot be overridden in subclasses. (Technically, double-equal can be implemented in subclasses, but the compiler ignores those implementors so they are never executed.) The implementation simply follows the two pointers to see if they point to the same address in memory. Even if two objects are alike in every possible way, if they're two different objects, they'll occupy two different locations in memory. Thus, they're copies: Changes to one are not reflected in the other. Yet it can be very difficult to tell them apart without changing them. The way to do this is to test if they're double-equal: Do they occupy the same location in memory?

Why is it important to know whether two handles point to the same object? Here's a simple example:

```
| set1 set2 |
set1 := Set withAll: #(a b c).
set2 := set1.
set2 add: #d.

Transcript cr; show: set1 printString.
Transcript cr; show: set2 printString.
```

When the Sets are displayed, set2 obviously contains #d, but does set1? Most new Smalltalkers would say that set1 does not contain #d because it was not sent "add: #d" like set2 was. The surprise is that set1 does contain #d just like set2 does because set1 and set2 are really two separate pointers to the same object. This code proves it:

```
| set1 set2 |
set1 := Set withAll: #(a b c).
set2 := set1.
Transcript cr; show: (set1 == set2) printString.
```

The result, true, shows that the two variables are really the same object. This is why when you change one of them, the other changes as well.

VisualWorks will actually show you an object's address, sort of, if you send it the message identityHash (previously called asOop). This returns a number that is unique for every object in the image. Thus two objects with the same address are really the same object and have the same identity-hash. This is helpful when you have two objects in two separate inspectors and you want to compare them to see if they're the same object. You can't very well use double-equal; which inspector would you run it in? But you can send identityHash to each object and visually compare the results.

WHAT IS EQUAL?

Although double-equal's implementation is clear and simple, equal's is anything but. Surprisingly, many developers do not seem to recognize this dilemma. Even for fairly oddball classes, programmers often think that implementing equal is perfectly straightforward. In one case, an experienced developer told me that if two views have the same model, they're equal (in his opinion). Since these two views might be two different kinds of widgets in two different windows, I had a hard time thinking of them as equivalent. Thus I find that what exactly makes two objects equal is often very subjective.

In VisualWorks, the most straightforward implementors of equal are in the ArithmeticValue hierarchy. Essentially, two ArithmeticValues (think of them as Numbers) are equal if the difference between them is zero. Similarly, if Dates or Times represent the same offset, they're equal. So the Magnitude hierarchy in general is pretty clear-cut.

When it comes to equal, the obvious answer is not necessarily so obvious once you think about it.

Then again, Characters are Magnitudes, but what does it mean for them to be equal? ParcPlace says that the two characters must be the exact same one. However, couldn't it be said that "A" is, in a sense, equal to "a"? Maybe, maybe not (which helps explain Character>> same As:). The point is not that ParcPlace is wrong, but that when it comes to equal, the obvious answer is not necessarily so obvious once you think about it.

Consider the Collection hierarchy, the second-most fertile source of implementors of equal. For two Collections to be equal, they have to contain the same number of elements and the two elements in each position have to be equal. This seems reasonable. However, this eliminates Sets because they are unordered; two equal elements being in the same position in the two Sets is just coincidence. It also means that #(a b c) does not equal #(c b a); although the elements are the same, the positions are different. Is it obvious that equal should work this way? Maybe, maybe not.

Most other classes that implement equal—and there aren't a whole lot of them—do so pretty unimaginatively. They verify that both objects are of the same class/species and that their instance variables have the same values. Thus a BlockClosure is only equal to another BlockClosure that has the same method, outer Context, and copiedValues. Does that really happen a lot?

EQUALITY AND TYPE

For two objects to be equal, they have to be of the same type. Objects which are not of the same type are not comparable, so they cannot be equal.

However, what constitutes a type in Smalltalk is unclear. For the purposes of determining equality, there are four main ways of determining if two objects are the same type:

- Are they instances of the same class? BlockClosure>>= (in VisualWorks) first checks that both objects are instances of the same class.
- Are they instances of the same species? SequenceableCollection>>= first verifies that the species of the two objects is the same.
- Are they instances from the same hierarchy? Interval>>= first confirms that the second object isKindOf: Interval.
- Do they claim to be the same type? String>>= initially confirms that the argument isString.

In each of these cases, equal makes sure that the two objects are of the same type by verifying that their classes are equivalent. This check is the first one made; if it fails, all other comparison is skipped.

Verifying that two objects are of the same type before making any other checks has an important benefit: equal can be used to compare any two objects and will not fail. If the argument is not of the right type, it will probably not understand the messages equal sends to it. This would cause a message-not-understood or similar error. Thus every implementor of equal should first verify that the receiver and argument are instances of comparable classes.

EQUAL AVOIDS DUPLICATES

If you want to know whether two objects *should* be equal, here's the question to ask yourself: Should I be able to store both objects in a Set, or should the second be considered a duplicate of the first? A Set is a collection that eliminates duplicates. So what constitutes a duplicate? If the object being added is equal to an object that is already in the collection, the Set does not add the new object. This eliminates duplicates.

If two objects are equal, they cannot be stored in a Set together. If they're unequal, a Set will not eliminate either as being a duplicate of the other. So you can forget defining equal through logical semantics and theoretical dissertations; the simplest answer is: Do you want to be able to store them both in a Set? Of course now programmers will argue the semantics of whether both objects should be able to live in the same Set.

This also explains what an IdentitySet is. Whereas a Set eliminates duplicate elements, an IdentitySet eliminates identical elements. Thus a Set tests for duplication using equal; an IdentitySet uses double-equal.

An IdentitySet is slightly more efficient than a Set because double-equal is faster than equal. However, using an IdentitySet to eliminate duplicates is dangerous. The IdentitySet will eliminate duplicates as long as equal and double-equal work the same for the objects in the collection. Yet this will fail if another developer implements equal for these objects so that it works differently than double-equal. Thus it's safest to eliminate duplicates using a Set. Use an IdentitySet to eliminate identical objects, or to store objects where duplicates are guaranteed to be identical objects (such as Symbols).

EQUAL IS DOUBLE-EQUAL

Because it is often not obvious what makes two objects equal, the default implementor of equal is

```
Object>>= anObject
    ^self == anObject
```

Thus for most objects, they're equal if (and only if) they're double-equal. If they don't have the same identity, they definitely don't have equality. Which is to say that equality doesn't make much sense for most objects. Identity always makes sense, which is why it's quite easy to design and implement. Equality, on the other hand, just doesn't make much sense for most types of objects.

You should avoid implementing equal arbitrarily. A standard protocol like equal is not very useful if its implementation is subjective and privy to the whim of the last person to implement it. If the user of your class can't tell what equal does without looking at your implementation, that method is not going to help him very much. In fact, it will hurt him if he's depending on all of the implementors in a hierarchy working polymorphically.

HASH

There is an easily overlooked but significant comment in VisualWorks' default implementor of equal:

```
Object>>= anObject
    "... If = is redefined in any subclass, consider also
     redefining the message hash."
```

This subtle suggestion is the only warning you get that equal and hash go hand-in-hand. This is because the chief user of hash (besides other implementors of hash) is findElementOrNil:. This is the method Set classes use to put an element in (add:) and find it again (includes: and remove:). For example, the implementor in Set contains these two lines:

```
Set>>findElementOrNil: anObject
    ...
    index := self initialIndexFor: anObject hash
boundedBy: length.
    [(probe := self basicAt: index) == nil or: [probe =
    anObject]] whileFalse: "keep looking"
    ...
```

Notice that anObject is being compared in two ways, hash and equal. This means that if two objects are equal, they need to hash to the same value. Otherwise, you could store an item and its equivalent together in the same Set.

For example, let's say you store "4.0" and "4" in a Set. Since they have the same hash value, the Set looks in 4.0's position, sees that it's equal to 4, and so doesn't store 4. If their hash values weren't equal, the Set would look in some other position, not find the 4.0, and store the 4 in the first empty slot it comes to. Similarly, if 4.0 is in a Set and you look for 4, you probably won't find it unless they have the same hash value.

By the way, the opposite is not true: two objects which hash to the same value do not need to be equal. A Set will avoid collisions more efficiently if unequal values have different hash values. Yet even if they do have the same hash value, the Set will handle the collision correctly because it realizes that they are not equal.

So whenever you implement equal, you should implement hash as well. If the implementor of hash is the same as its super-implementor, you don't need the new one, but it's important that you at least thought about it.

CONCLUSIONS

Here are the eight main points in this article:

- A handle is a pointer to an object. A variable is a handle.
- Everything is Smalltalk is a pointer to an object.
- Identity means that two handles hold the same object.
- Equality means that two handles hold equivalent objects, but equivalency is fairly subjective.
- For two objects to be equal, they have to at least be of the same type. Each implementor of equal should first check that both objects are of the same type.
- A Set considers two objects duplicates if they are equal. Duplicates in an IdentitySet are identical.
- Equality doesn't make sense for most types of objects, in which case equality is the same as identity.
- Two objects which are equal need to have the same hash value.

In my next article, I'll talk about three different types of instance variables. One of these types is instrumental in determining object equality.

Bobby Woolf is a Senior Member of Technical Staff at Knowledge Systems Corp. in Cary, NC. He mentors Smalltalk developers in the use of VisualWorks, ENVY, and Design Patterns. He welcomes your comments at woolf@acm.org or at http://www.ksccary.com.