

Lecture 12: Equality and Copying

October 2, 2002

12.1 The Object Contract

Every class extends `Object`, and therefore inherits all of its methods. Two of these are particularly important and consequential in all programs; the method for testing equality:

```
public boolean equals (Object o)
```

and the method for generating a hash code:

```
public int hashCode ()
```

Like any other methods of a superclass, these methods can be overridden. We'll see in a later lecture on subtyping that a subclass should be a *subtype*. That means that it should behave according to the specification of the superclass, so that an object of the subclass can be placed in a context in which a superclass object is expected, and still behave appropriately.

The specification of the `Object` class is rather abstract and may seem abstruse. But failing to obey it has dire consequences, and tends to result in horrible obscure bugs. Worse, if you don't understand this specification and its ramifications, you are likely to introduce flaws in your code that have a pervasive effect and are hard to eliminate without major reworking. The specification of the `Object` class is so important that it is often referred to as *the Object contract*.

The contract can be found in the method specifications for `equals` and `hashCode` in the Java API documentation. It states that:

- `equals` must define an equivalence relation – that is, a relation that is reflexive, symmetric, and transitive;
- `equals` must be consistent: repeated calls to the method must yield the same result unless the arguments are modified in between;
- for a non-null reference `x`, `x.equals (null)` should return `false`; and
- `hashCode` must produce the same result for two objects that are deemed equal by the `equals` method.

12.2 Equality Properties

Let's look first at the properties of the `equals` method. Reflexivity means that an object always equals itself; symmetry means that when `a.equals b`, `b.equals a`; transitivity means that when `a.equals b` and `b.equals c`, `a` also equals `c`.

These may seem like obvious properties, and indeed they are. If they did not hold, it's hard to imagine how the `equals` method would be used: you'd have to worry about whether to write `a.equals(b)` or `b.equals(a)`, for example, if it weren't symmetric.

What is much less obvious, however, is how easy it is to break these properties inadvertently. The following example (taken from Joshua Bloch's excellent *Effective Java: Programming Language Guide*, one of the course recommended texts) shows how symmetry and transitivity can be broken in the presence of inheritance.

Consider a simple class that implements a two-dimensional point:

```

public class Point {
    private final int x;
    private final int y;
    public Point (int x, int y) {
        this.x = x; this.y = y;
    }
    public boolean equals (Object o) {
        if (!(o instanceof Point))
            return false;
        Point p = (Point) o;
        return p.x == x && p.y == y;
    }
    ...
}

```

Now suppose we add the notion of a color:

```

public class ColorPoint extends Point {
    private Color color;
    public ColorPoint (int x, int y, Color color) {
        super (x, y);
        this.color = color;
    }
    ...
}

```

What should the equals method of ColorPoint look like? We could just inherit equals from Point, but then two ColorPoints will be deemed equal even if they have different colors. We could override it like this:

```

public boolean equals (Object o) {
    if (!(o instanceof ColorPoint))
        return false;
    ColorPoint cp = (ColorPoint) o;
    return super.equals (o) && cp.color.equals (color);
}

```

This seemingly inoffensive method actually violates the requirement of symmetry. To see why, consider a point and a color point:

```

Point p = new Point (1, 2);
ColorPoint cp = new ColorPoint (1, 2, Color.RED);

```

Now `p.equals(cp)` will return true, but `cp.equals(p)` will return false! The problem is that these two expressions use different equals methods: the first uses the method from Point, which ignores color, and the second uses the method from ColorPoint.

We could try to fix this by having the equals method of ColorPoint ignore color when comparing against a non-colored point:

```

public boolean equals (Object o) {
    if (!(o instanceof Point))
        return false;
    // if o is a normal Point, do a color-blind compare

```

```

    if (!(o instanceof ColorPoint))
        return super.equals (o);
    ColorPoint cp = (ColorPoint) o;
    return super.equals (o) && cp.color.equals (color);
}

```

This solves the symmetry problem, but now equality isn't transitive! To see why, consider constructing these points:

```

ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point (1, 2);
ColorPoint p3 = new ColorPoint (1, 2, Color.BLUE);

```

The calls `p1.equals(p2)` and `p2.equals(p3)` will both return true, but `p1.equals(p3)` will return false.

It turns out there is no solution to this problem: it's a fundamental flaw in inheritance. You can't write a good equals method for `ColorPoint` if it inherits from `Point` as written. However, there are two workarounds. The first is to change `Point`'s equals method so that it rejects equality with *any* of its subclasses:

```

public class Point {
    ...
    public boolean equals (Object o) {
        if (o == null || !o.getClass().equals(getClass ()))
            return false;
        Point p = (Point) o;
        return p.x == x && p.y == y;
    }
    ...
}

```

Explicit comparison of classes is a stronger test than `instanceof`. A `ColorPoint` is an instance of `Point`, but `ColorPoint.getClass() != Point.getClass ()`. The drawback of this approach is that you lose the ability for harmless `Point` subclasses to compare equal to a `Point`. For example, you might write a subclass `PolarPoint` that doesn't add any new attributes to `Point`, but merely offers some new methods for obtaining the point in polar coordinates. With the `getClass` workaround, a `PolarPoint` can never equal a `Point`, even though it has the same x,y value internally.

The second workaround is to implement `ColorPoint` using `Point` in its representation, rather than inheriting it:

```

public class ColorPoint {
    Point pt;
    Color color;
    ...
}

```

Since `ColorPoint` no longer extends `Point`, the problem goes away. See Bloch's book for more details about this approach.

Bloch's book also give some hints on how to write a good equals method, and he points out some common pitfalls. For example, what happens if you write something like this:

```

public boolean equals (Point p)

```

where you've substituted another type for `Object` in the declaration of equals?

12.3 Equality for Immutable and Mutable Types

Although equals is required to define an equivalence relation, the specification leaves unanswered the important question of *which* equivalence relation equals should define. When should two objects be considered equals?

For immutable types, the answer is straightforward: two immutable objects are equal when they represent the same abstract value. Thus, two Strings are equal when they represent the same sequence of characters.

Mutability makes this issue much more subtle. Suppose I have two empty Lists:

```
List t1 = new ArrayList ();  
List t2 = new ArrayList ();
```

These lists represent the same abstract value, an empty sequence, so from one point of view you might consider them equal. But they are different in the sense that I can mutate one of them, say by adding an element, without affecting the other, so that the two lists subsequently represent different abstract values.

These two perspectives are formalized by Professor Liskov in the course text book as *observational equivalence* and *behavioral equivalence*. Two objects are observationally equivalent if they cannot be distinguished by calling any sequence of observer methods on them. Two objects are behaviorally equivalent if they cannot be distinguished by *any* sequence of methods, including mutators.

Obviously, for immutable types, observational equivalence and behavioral equivalence mean the same thing, because immutable types have no mutators. For mutable types, we can choose to define equals as either behavioral or observational equivalence. Each approach has its advantages and drawbacks.

12.3.1 Liskov Approach

In the course text, Professor Liskov defines *equals* as behavioral equivalence, and introduces a new method named *similar* to represent observational equivalence. Here's how you code *equals* and *similar*. For a mutable type, you simply inherit the *equals* method from Object, but you write a *similar* method that performs an element-by-element comparison. For an immutable type, you override *equals* with a method that performs an element-by-element comparison, and you have *similar* call *equals* so that they are the same.

This approach, when applied uniformly, is easy to understand and works well. But it's not always ideal. For example, it prevents you from using mutable lists as values. Suppose you're writing a program for registering students in courses. The prerequisites of a course might be a set of permitted course sequences, with each sequence represented as a List like [6.001, 6.002] or [6.045, 6.046]. When a student comes along and presents a valid course sequence, which has been constructed as a separate List object, you will be unable to find it in your set of prerequisites. The equals test fails because the student's query is not behaviorally equivalent (i.e., not the same object) as any List in your set.

12.3.2 Java Collections Approach

For reasons such as this, the designer of the Java Collections API did not follow this approach. There is no *similar* method, and *equals* is defined as observational equivalence.

This has some convenient consequences, but it also has some unfortunate ones. The worst is the risk of rep exposure. Suppose part of Set's rep invariant is the requirement that it contain no duplicates, i.e., that for all $i \neq j$, elt[i].equals(elt[j]) always returns false. If *equals* is defined as observational equivalence, then a client can mutate an object in the set to make it equal to another object in the set, thereby breaking Set's rep invariant. The Liskov approach doesn't suffer from this problem. Behavioral equivalence ensures that if two objects are equal or unequal at one point, they continue to be equal or unequal forever after; there is *nothing* you can do to two unequal objects that will make them become equal.

This leaves you with two choices, both of which are acceptable in 6.170:

- You can follow the Java approach, in which case you'll get the benefits of its convenience, but you'll have to deal with the complications that can arise.
- Alternatively, you can follow the Liskov approach, but in that case you'll need to figure out how to incorporate into your code the Java collection classes, such as List and Set.

In general, when you have to incorporate a class whose *equals* method follows a different approach from the program as a whole, you can write a wrapper around the class that replaces the *equals* method with a more suitable one. The course text gives an example of how to do this.

12.4 Hashing

Now we come to the part of the contract relating to the `hashCode` method. To understand it, you'll need to have some idea of how hash tables work.

Hashtables are a fantastic invention one of the best ideas of computer science. A hashtable is a representation for a mapping: an abstract data type that maps keys to values. Hashtables offer constant time lookup, so they tend to perform better than trees or lists. Keys don't have to be ordered, or have any particular property, except for offering *equals* and *hashCode*.

Here's how a hashtable works. It contains an array that is initialized to a size corresponding to the number of elements that we expect to be inserted. When a key and a value are presented for insertion, we compute the hashcode of the key, and convert it into an index in the array's range (eg, by a modulo division). The value is then inserted at that index.

The rep invariant of a hash table includes the fundamental constraint that keys are in the slots determined by their hash codes. We'll see later why this is important.

Hashcodes are designed so that the keys will be spread evenly over the indices. But occasionally a conflict occurs, and two keys are placed at the same index. So rather than holding a single value at an index, a hashtable actually holds a list of key/value pairs (usually called hash buckets), implemented in Java as objects from class `HashMap` with two fields. On insertion, you add a pair to the list in the array slot determined by the hash code. For lookup, you hash the key, find the right slot, and then examine each of the pairs until one is found whose key matches the given key.

Now it should be clear why the Object contract requires equal objects to have the same hash key. If two equal objects had distinct hash keys, they might be placed in different slots. So if you attempt to lookup a value using a key equal to the one with which it was inserted, the lookup may fail.

A simple and drastic way to ensure that the contract is met is for `hashCode` to always return some constant value, so every object's hash code is the same. This satisfies the Object contract, but it would have a disastrous performance effect, since every key will be stored in the same slot, and every lookup will degenerate to a linear search along a long list.

The standard way to construct a more reasonable hash code that still satisfies the contract is to compute a hash code for each component of the object that is used in the determination of equality (usually by calling the `hashCode` method of each component), and then combining these, throwing in a few arithmetic operations. Look at Bloch's book for details.

Most crucially, note that if you don't override `hashCode` at all, you'll get the one from Object, which is based on the address of the object. If you have overridden *equals*, this will mean that you will have almost certainly violated the contract. So as a general rule:

Always override hashCode when you override equals.

(This is one of Bloch's aphorisms. The whole book is a collection of aphorisms like it, each nicely explained and illustrated.)

One year, a 6.170 student spent hours tracking down a bug in a project that amounted to nothing more than misspelling `hashCode` as `hashcode`. This created a method that didn't override the `hashCode` method of Object at all, and strange things happened. Another reason to avoid inheritance...

12.5 Copying

Let's look back at the rep exposure problem we discussed two sections ago. Using the Liskov approach, defining equals as behavioral equivalence, doesn't solve all the problems. Any time we have a collection of arbitrary objects where the rep invariant of the collection depends on some property of the objects, we have the potential for rep exposure.

Consider a SortedList data type whose rep invariant requires that its elements be kept in sorted order. (The elements are required to implement the Comparable interface, so that SortedList can compare them by calling compareTo). If the elements are mutable in such a way that their sort order can be changed, then SortedList is also exposing its rep. It isn't at all clear how to fix compareTo to prevent the problem in the same way that defining equals as behavioral equivalence prevented it above.

One approach to this problem is *defensive copying*, where SortedList makes a private copy of every element added to the list. To keep its elements private, SortedList must also make copies when it hands elements back in response to lookups. This approach would solve the rep exposure, but unfortunately it can't be implemented in Java. You might think you can copy an arbitrary object by calling its clone method, which is defined in Object:

```
protected Object clone () throws CloneNotSupportedException
```

But as the declaration of this method suggests, implementing clone is optional, and it isn't even available as a public method unless a class overrides it and redeclares it public. Thus, *there is no way to copy an arbitrary Object in Java*.

The clone method is very tricky to implement, anyway, so I recommend that you don't use it at all unless you have to. See Bloch's book for an excellent discussion of the problems and pitfalls of clone.

So what should you do instead? If you know the type of the object you want to copy, the most common approach is a *copy constructor*:

```
public Point (Point pt) {  
    x=pt.x; y=pt.y;  
}
```

The copy constructor approach is widely used in the Java API. A nice feature of this approach is that it allows the client to choose the class of the object to be created. The argument to the copy constructor is often declared to have the type of an interface so that you can pass the constructor any type of object that implements the interface. All of Java's collection classes, for example, provide a copy constructor that takes an argument of type Collection or Map. If you want to create an array list from a linked list *l1ist*, for example, you would just call

```
new ArrayList(l1ist)
```

12.6 Summary

Issues of copying and equality show the power of object-oriented programming but also its pitfalls. You must have a systematic and uniform treatment of equality, hashing and copying in any program you write.