Lecture 9: Equality, Copying and Views

9.1 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*. This 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;
- *hashCode* must produce the same result for two objects that are deemed equal by the *equals* method;

9.2 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 seems 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 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 colour:

```
public class ColourPoint extends Point {
    private Colour colour;
    public ColourPoint (int x, int y, Colour colour) {
        super (x, y);
        this.colour = colour;
        }
    ...
}
```

What should the *equals* method of *ColourPoint* look like? We could just inherit *equals*

from *Point*, but then two *ColourPoints* will be deemed equal even if they have different colours. We could override it like this:

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

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

```
Point p = new Point (1, 2);
ColourPoint cp = new ColourPoint (1, 2, Colour.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 colour, and the second uses the method from *ColourPoint*.

We could try and fix this by having the equals method of *ColourPoint* ignore colour when comparing to a non-colour point:

```
public boolean equals (Object o) {
    if (!(o instanceof Point))
        return false;
    // if o is a normal Point, do colour-blind comparison
    if (!(o instanceof ColourPoint))
        return o.equals (this);
    ColourPoint cp = (ColourPoint) o;
    return super.equals (o) && cp.colour.equals (colour);
    }
```

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

```
ColourPoint p1 = new ColourPoint (1, 2, Colour.RED);
Point p2 = new Point (1, 2);
ColourPoint p2 = new ColourPoint (1, 2, Colour.BLUE);
```

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

It turns out that there is no solution to this problem: it's a fundamental problem of inheritance. You can't write a good *equals* method for *ColourPoint* if it inherits from *Point*. However, if you implement *ColourPoint* using *Point* in its representation, so that a *ColourPoint* is no longer treated as a *Point*, the problem goes away. See Bloch's book for details.

Bloch's book also gives 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)

substituting another type for *Object* in the declaration of *equals*?

9.3 Hashing

To understand the part of the contract relating to the *hashCode* method, 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 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 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.)

Last year, a student spent hours tracking down a bug in a project that amounted to nothing more than mispelling *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…

9.4 Copying

The need to make a copy of an object often arises. For example, you may want to do a computation that requires modifying the object but without affecting objects that already hold references to it. Or you may have a 'prototype' object that you want to make a collection of objects from that differ in small ways, and it's convenient to make copies and then modify them.

People sometimes talk about 'shallow' and 'deep' copies. A shallow copy of an object is made by creating a new object whose fields point to the same objects as the old object. A deep copy is made by creating a new object also for the objects pointed to by the fields, and perhaps for the objects they point to, and so on.

How should copying be done? If you've diligently studied the Java API, you may assume that you should use the *clone* method of *Object*, along with the *Cloneable* interface. This is tempting, because *Cloneable* is a special kind of 'marker interface' that adds functionality magically to a class. Unfortunately, though, the design of this part of Java isn't quite right, and it's very difficult to use it well. So I recommend that you don't use it at all, unless you have to (eg, because code you're using requires your class to implement *Cloneable*, or because your manager hasn't taken 6170). See Bloch's book for an insightful discussion of the problems.

You might think it would be fine to declare a method like this:

```
class Point {
    Point copy () {
    ...
        }
    ...
    }
```

Note the return type: copying a point should result in a point. Now in a subclass, you'd like the *copy* method to return a subclass object:

```
class ColourPoint extends Point {
    ColourPoint copy () {
    ...
        }
    ...
    }
```

Unfortunately, this is not legal in Java. You can't change the return type of a method when you override it in a subclass. And overloading of method names uses only the types of the arguments. So you'd be forced to declare both methods like this:

```
Object copy ()
```

and this is a nuisance, because you'll have to downcast the result. But it's workable and sometimes the right thing to do.

There are two other ways to do copying. One is to use a static method called a 'factory' method because it creates new objects:

```
public static Point newPoint (Point p)
```

The other is to provide additional constructors, usually called 'copy constructors':

public Point (Point p)

Both of these work nicely, although they're not perfect. You can't put static methods or constructors in an interface, so they're not when you're trying to provide generic functionality. 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 *l*, for example, you would just call

new ArrayList (l)

9.5 Element and Container Equality

When are two containers equal? If they are immutable, they should be equal if they contain the same elements. For example, two strings should be equal if they contain the same characters (in the same order). Otherwise, if we just kept the default *Object* equality method, a string entered at the keyboard, for example, would never match a string in a list or table, because it would be a new string object and therefore not the same object as any other. And indeed, this is exactly how *equals* is implemented in the Java *String* class, and if you want to see if two strings *s1* and *s2* contain the same character sequence, you should write

s1.equals (s2)

and not

s1 == s2

which will return false when *s1* and *s2* denote different string objects that contain the same character sequences.

9.5.1 The Problem

So much for strings – sequences of characters. Let's consider lists now, which are sequences of arbitrary objects. Should they be treated the same way, so that two lists are equal if they contain the same elements in the same order?

Suppose I'm planning a party at which my friends will sit at several different tables, and

I've written a program to help me create a seating plan. I represent each table as a list of friends, and the party as a whole as a set of these lists. The program starts by creating empty lists for the tables and inserting them into the set:

```
List t1 = new LinkedList ();
List t2 = new LinkedList ();
...
Set s = new HashSet ();
s.add (t1);
s.add (t2);
...
```

At some later point, the program will add friends to the various lists; it may also create new lists and replace existing lists in the set with them. Finally, it iterates over the contents of the set, printing out each list.

This program will fail, because the initial insertions will not have the expected effect. Even though the empty lists represent conceptually distinct table plans, they will be equal according to the *equals* method of *LinkedList*. Since *Set* uses the *equals* method on its elements to reject duplicates, all insertions but the first will have no effect, since all of the empty lists will be deemed duplicates.

How can we solve this problem? You might think that *Set* should have used == to check for duplicates instead, so that an object is regarded as a duplicate only if that very object is already in the set. But that wouldn't work for strings; it would mean that after

```
Set set = new HashSet ();
String lower = "hello";
String upper = "HELLO";
set.add (lower.toUpperCase());
...
```

the test *set.contains (upper)* would evaluate to false, since the *toUpperCase* method creates a new string.

9.5.2 The Liskov Solution

In our course text, Professor Liskov presents a systematic solution to this problem. You provide two distinct methods: *equals*, which returns true when two objects in a class are behaviourally equivalently, and *similar*, which returns true when two objects are observationally equivalent.

Here's the difference. Two objects are *behaviourally equivalent* if there is no sequence of operations that can distinguish them. On these grounds, the empty lists *t1* and *t2* from above are not equivalent, since if you insert an element into one, you can see that the other doesn't change. But two distinct strings that contain the same sequence of characters are equivalent, since you can't modify them and thus discover that they are different objects. (We're assuming you're not allowed to use == in this experiment.)

Two objects are *observationally equivalent* if you can't tell the difference between them using observer operations (and no mutations). On these grounds, the empty lists *t1* and *t2* from above are equivalent, since they have the same size, contain the same elements, etc. And two distinct strings that contain the same sequence of characters are also equivalent.

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 a field-by-field comparison. For an immutable type, you override *equals* with a method that performs a field-by-field comparison, and have *similar* call *equals* so that they are the same.

This solution, when applied uniformly, is easy to understand and works well. But it's not always ideal. Suppose you want to write some code that interns objects. This means mutating a data structure so that references to objects that are structurally identical become references to the very same object. This is often done in compilers; the objects might be program variables, for example, and you want to have all references to a particular variable in the abstract syntax tree point to the same object, so that any information you store about the variable (by mutating the object) is effectively propagated to all sites in which it appears.

To do the interning, you might try to use a hash table. Every time you encounter a new object in the data structure, you look that object up in the table to see if it has a canonical representative. If it does, you replace it by the representative; otherwise you insert it as both key and value into the table.

Under the Liskov approach, this strategy would fail, because the equality test on the keys of the table would never find a match for distinct objects that are structurally equivalent, since the *equals* method of a mutable object only returns true on the very same object.

9.5.3 The Java 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 observational equivalence.

This has some convenient consequences. The interning table will work, for example. But it also has some unfortunate consequences. The seating plan program will break, because two distinct empty lists will be deemed equal.

In the Java *List* specification, two lists are equal not only if they contain the same elements in the same order, but also if they contain *equal* elements in the same order. In other words, the *equals* method is called recursively. To maintain the *Object* contract, the *hashCode* method is also called recursively on the elements. This results in a very nasty surprise. The following code, in which a list is inserted into itself, will actually fail to terminate!

```
List l = new LinkedList ();
l.add (l);
int h = l.hashCode ();
```

This is why you'll find warnings in the Java API documentation about inserting containers into themselves, such as this comment in the specification of *List*:

> *Note: While it is permissible for lists to contain themselves as elements, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a list.*

There are some other, even more subtle, consequences of the Java approach to do with rep exposure which are explained below.

This leaves you with two choices, both of which are acceptable in 6170:

- 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 *LinkedList* and *HashSet*).

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.

9.6 Rep Exposure

Let's revisit the example of rep exposure that we closed yesterday's lecture with. We imagined a variant of *LinkedList* for representing sequences without duplicates. The *add* operation has a new specification saying that the element is added only if it isn't a duplicate, and its code performs this check:

```
void add (Object o) {
    if (contains (o))
        return;
    else
        // add the element
        ...
    }
```

We record the rep invariant at the top of the file saying that the list contains no duplicates:

> *The list contains no duplicates. That is, there are no distinct entries e1 and e2 such that e1.element.equals (e2.element).*

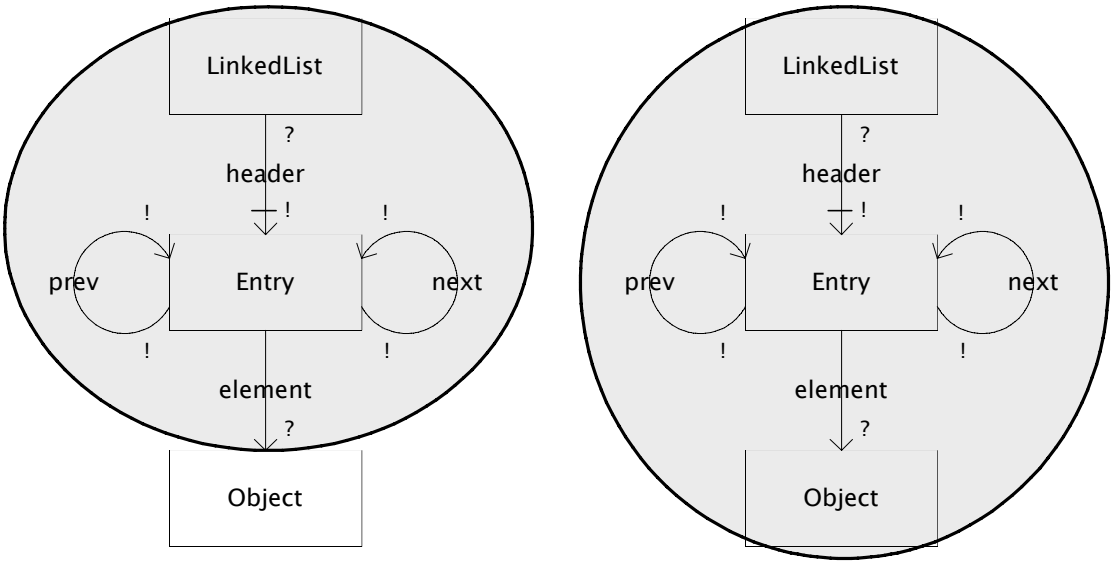
We check that it's preserved, by ensuring that every method that adds an element first performs the containment check.

Unfortunately, this isn't good enough. Watch what happens if we make a list of lists and then mutate an element list:

```
List x = new LinkedList ();
List y = new LinkedList ();
Object o = new Object ();
x.add (o);
List p = new LinkedList ();
p.add (x);
p.add (y);
x.remove (o);
```

After this code sequence, the rep invariant of p is broken. The problem is that the mutation to x makes it equal to y, since they are then both empty lists.

What's going on here? The contour that we drew around the representation actually includes the element class, since the rep invariant depends on a property of the element (see figure). Note that this problem would not have arisen if equality had been determined by the Liskov approach, since two mutable elements would be equal only if they were the very same object: the contour extends only to the reference to the element, and not to the element itself.



9.6.1 Mutating Hash Keys

A more common and insidious example of this phenomenon occurs with hash keys. If you mutate an object after it has been inserted as a key in a hash table, its hash code may change. As a result, the crucial rep invariant of the hash table – that keys are in the slots determined by their hash codes – is broken.

Here's an example. A hash set is a set implemented with a hash table: think of it as a hash table with keys and no values. If we insert an empty list into a hash set, and then add an element to the list like this:

Set s = new HashSet ();
List x = new LinkedList ();
s.add (x);
x.add (new Object ());

a subsequent call *s.contains(x)* is likely to return *false*. If you think that's acceptable, consider the fact that there may now be *no value of x* for which *s.contains(x)* returns true, even though *s.size()* will return 1!

Again, the problem is rep exposure: the contour around the hash table includes the keys.

The lesson from this is: either follow the Liskov approach, and wrap the Java list to

override its *equals* method, or make sure that you never mutate hash keys, or allow any mutation of an element of a container that might break the container's rep invariant. This is why you'll see comments such as this in the Java API specification:

> *Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element.*

9.7 Views

An increasingly common idiom in object-oriented program is to have distinct objects that offer different kinds of access to the same underlying data structure. Such objects are called *views*. Usually one object is thought of as primary, and another as secondary. The primary one is called the 'underlying' or 'backing' object, and the secondary one is called the 'view'.

We are used to aliasing, when two object references point to the same object, so that a change under one name appears as a change under the other:

```
List x = new LinkedList ();
List y = x;
y.add (o); // changes y also
```

Views are tricky because they involve a subtle form of aliasing, in which the two objects have distinct types. We have seen an example of this with iterators, whose *remove* method of an iterator removes the last yielded element from the underlying collection:

```
List x = new LinkedList ();
...
Iterator i = x.iterator ();
while (i.hasNext ()) {
    Object o = i.next ();
    ...
    i.remove (); // mutates x also
    }
```

An iterator is this a view on the underlying collection. Here are two other examples of views in the Java collections API.

- Implementations of the *Map* interface are required to have a method *keySet* which returns the set of keys in the map. This set is a view; as the underlying map changes, the set will change accordingly. Unlike an iterator, this view and the underlying map can be both be modified; a deletion of a key from the set will cause the key and its value to be deleted from the map. The set does not support an *add* operation, since it would not make sense to add a key without a value. (This, by the way, is why *add* and *remove* are optional methods in the *Set* interface.)
- *List* has a method *subList* which returns a view of part of a list. It can be used to access the list with an offset, eliminating the need for explicit range operations. Any operation that expects a list can be used as a range operation by passing a *subList* view instead of a whole list. For example, the following idiom removes a range of elements from a list:

  list.subList(from, to).clear();

Ideally, a view and its backing object should both be modifiable, with effects propagated as expected between the two. Unfortunately, this is not always possible, and many views place constraints on what kinds of modification are possible. An iterator, for example, becomes invalid if the underlying collection is modified during iteration. And a sublist is invalidated by certain structural modifications to the underlying list.

Things get even trickier when there are several views on the same object. For example, if you have two iterators simultaneously on the same underlying collection, a modification through one iterator (by a call to *remove*) will invalidate the other iterator (but not the collection).

9.8 Summary

Issues of copying, views 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. Views are a very useful mechanism, but they must be handled carefully. Constructing an object model of your program is useful because it will remind you of where sharings occur and cause you to examine each case carefully.