Lecture 12 : Identity and Equality I

13.1 Quote of the Day

I like pigs. Dogs look up to us. Cats look down on us. Pigs treat us as equals.

Winston Churchill

13.2 Context

What you'll learn: Object equality and hash codes; how to meet the object contract.

Why you should learn this: As your objects are shuffled around in your program, into and out of collections for example, they will from time to time be tested for equality; if you haven't thought carefully about what this means for your classes, the result will be bugs whose effects are very non-local and hard to trace.

What I assume you already know: Proficient in reading specifications, somewhat familiar with the Java collection classes and the use of hash maps.

13.3 The Object Contract

In Java, 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, and the method for generating a hash code.

```
public boolean equals(Object o);
public int hashCode();
```

Like any other methods of a superclass, these methods can be overridden. We will see in lecture next week 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. So a class that overrides `equals` and `hashCode` should scrupulously obey their specification.

The specification of the `Object` class given in its documentation is rather abstract and may seem abstruse. But failing to obey it has dire consequences, and tends to result in horribly obscure bugs. Worse, if you do not 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 – i.e. be 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.

13.4 Equality and Inheritance

For the moment, let us ignore the `hashCode` method and 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 is hard to imagine how the equals method would be used: you would have to worry about whether to write `a.equals(b)` or `b.equals(a)`, for example, if it were not symmetric.

What is much less obvious, however, is how easy it is to break these properties inadvertently. The following example shows how symmetry and transitivity can be broken in the presence of inheritance. Consider a simple class that stores a duration in time, with a field for the number of days and the number of seconds:

```
public class Duration {
    private final int day;
    private final int sec;
    public Duration(int day, int sec) {
        this.day = day;  this.sec = sec;
    }
    public boolean equals(Object o) {
        if (!(o instanceof Duration))
            return false;
        Duration d = (Duration) o;
        return d.day == day && d.sec == sec;
    }
}
```

The `equals` method takes an `Object` as its argument; this is mandated by the object contract. The method returns `true` if the object passed is a `Duration` and stores the same interval (assume for now that the `day` and `sec` fields need to be exactly the same for two `Duration` objects to be considered equal). Now suppose find that we sometimes have a need for more precision, and we add a field for nanoseconds in a derived class:

```
public class NanoDuration extends Duration {
    private final int nano;
    public NanoDuration(int day, int sec, int nano) {
        super(day,sec);
        this.nano = nano;
    }
}
```

What should the `equals` method of `NanoDuration` look like? We could just inherit `equals` from `Duration`, but then two `NanoDurations` will be deemed equal even if they do indeed differ by nanoseconds. We could override it like this:

```
public boolean equals(Object o) {
    if (!(o instanceof NanoDuration))
        return false;
    NanoDuration nd = (NanoDuration) o;
    return super.equals(nd) && nano == nd.nano;
}
```

This seemingly inoffensive method actually violates the requirement of symmetry. To see why, consider a `Duration` and a `NanoDuration`:

```
Duration d = new Duration(1,12);
NanoDuration nd = new NanoDuration(1,12,123);
System.out.println(d.equals(nd)); // true
System.out.println(nd.equals(d)); // false! – not symmetric
```

Notice that `d.equals(nd)` returns true, but `nd.equals(d)` returns false! The problem is that these two expressions use different `equals` methods: the first uses the method from `Duration`, which ignores nanoseconds, and the second uses the method from `NanoDuration`.

We could try to fix this by having the `equals` method of `NanoDuration` ignore nanoseconds when comparing against a normal `Duration`:

```
public boolean equals(Object o) {
    if (!(o instanceof Duration))
        return false;
    // if o is a normal Duration, compare without nano field
    if (!(o instanceof NanoDuration))
        return super.equals(o);
    NanoDuration nd = (NanoDuration) o;
    return super.equals(nd) && nano == nd.nano;
}
```

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

```
NanoDuration d1 = new NanoDuration(1,12,123);
Duration d2 = new Duration(1,12);
NanoDuration d3 = new NanoDuration(1,12,999);
System.out.println(d1.equals(d2)); // true
System.out.println(d2.equals(d3)); // true
System.out.println(d1.equals(d3)); // false! – not transitive
```

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 is a fundamental flaw in inheritance. You cannot write a good `equals` method for `NanoDuration` if it inherits from `Duration` as written. However, there are two workarounds. The first is to change `Duration`'s `equals` method so that it rejects equality with any of its subclasses:

```
public class Duration {
    //...
    public boolean equals(Object o) {
        if (o == null || !o.getClass().equals(getClass()))
            return false;
        Duration d = (Duration) o;
        return d.day == day && d.sec == sec;
    }
    //...
}
```

Explicit comparison of classes is a stronger test than `instanceof`. A `NanoDuration` is an `instanceof` `Duration`, but `NanoDuration.getClass() != Duration.getClass()`. The drawback of this approach is that you lose the ability for harmless `Duration` subclasses to compare for equality to a `Duration`. For example, you might write a subclass `ArithmeticDuration` that doesn't add any new attributes to `Duration`, but merely offers some new methods for adding and subtracting durations. With the `getClass` workaround, a `ArithmeticDuration` can never equal a `Duration`, even though it has the same `(day,sec)` value internally. The second workaround is to implement `NanoDuration` using `Duration` in its representation, rather than inheriting it:

```
public class NanoDuration {
    final Duration d;
    final int nano;
    //...
}
```

Since `NanoDuration` no longer extends `Duration`, the problem goes away. This strategy is called *composition*; we will see more of it in a later lecture. Bloch's book 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(Duration d)
```

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

The problems we've discussed here are based on two Java classes, `Date` and `Timestamp`. `Timestamp` extends `Date` by adding a nanosecond field to it, just like our example. This causes so many problems when objects of the two classes are compared that more recent versions of the specification for `Timestamp` are full of caveats and complexities:

> Note: This type is a composite of a java.util.Date and a separate nanoseconds value. Only integral seconds are stored in the java.util.Date component. The fractional seconds - the nanos - are separate. The Timestamp.equals(Object) method never returns true

> when passed a value of type java.util.Date because the nanos component of a date is unknown. As a result, the Timestamp.equals(Object) method is not symmetric with respect to the java.util.Date.equals(Object) method. Also, the hashcode method uses the underlying java.util.Data implementation and therefore does not include nanos in its computation.
>
> Due to the differences between the Timestamp class and the java.util.Date class mentioned above, it is recommended that code not view Timestamp values generically as an instance of java.util.Date. The inheritance relationship between Timestamp and java.util.Date really denotes implementation inheritance, and not type inheritance.

In this last paragraph, the designers are basically throwing up their hands and saying that the only way to solve this problem is to deny `Timestamp`'s parentage, and treat it as an entirely separate class to `Date`. In retrospect, this could have been done by using `Date` in the *implementation* of `Timestamp`, rather than using it in the specification. We will return to this topic in a later lecture.

13.5 Equality and Efficiency

To understand the part of the Object contract relating to the `hashCode` method, you'll need to have some idea of how hash tables work. Hash tables are a fantastic invention – one of the best ideas of computer science. A hash table is a representation for a mapping: an abstract data type that maps keys to values. Hash tables 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 hash table 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 (e.g., by a modulo division). The value is then inserted at that index.

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 hash table 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 Joshua Bloch's book *Effective Java* 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.)

Some years ago, so the legend goes, a 6.170 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...

What hash code would work for our `Duration` class, where `a.equals(b)` if they both have the same `day` and `sec` fields? Here are some possibilities that would work, but lead to greater or lesser efficiency in hash tables:

```
// always safe, but makes hash tables completely inefficient
public int hashCode() {
    return 1;
}
 // safe; but collisions for Durations that differ in sec only
public int hashCode() {
    return day;
}
 // safe; collisions possible but don't seem likely to happen systematically
public int hashCode() {
    return day+sec;
}
```

It is important to realize that the default `hashCode` implementation inherited from `Object` is *not* appropriate. The default implementation is generally based on some part of the address of an object in memory. This works just fine as a hash code if the only thing an object is equal to is itself. But once we can create two objects which we wish to treat as equal (for example, two `Duration` objects with the same `day` and `sec` fields), then this hash code is inappropriate, since it may give these equal objects different hash codes.

Suppose we now changed our idea of equality for `Duration` to the following:

```
public boolean equals(Object o) {
    if (!(o instanceof Duration))
        return false;
    Duration d = (Duration) o;
    return 24*60*60*day+sec == 24*60*60*d.day+d.sec;
}
```

Now the `day` and `sec` fields do not have to be exactly the same for two `Duration`s to be equal, as long as the total number of seconds represented is the same. Hash codes of `day` or `day+sec` as we had above now no longer work, since two equal `Duration`s might be given different hash codes. Here is one that works:

```
public int hashCode() {
    return 24*60*60*day + sec;
}
```

So we see that whenever the behavior of the `equals` method changes, we must reconsider whether the `hashCode` method is still safe.