1 The Object Contract

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

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

Like any other methods of a superclass, the `equals` methods can be overridden. We have seen in the lecture on subclassing that we should be careful when using inheritance. Subclass objects 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 the `Object` class. Here we will focus on the contract for `equals`. When you override the `equals` method, you must adhere to its general contract. 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 provided no information used in `equals` comparisons on the object is modified;
- 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.

2 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. To keep this lecture self-contained, we describe hash tables below.

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.

The rep invariant of a hash table includes the fundamental constraint that keys are in the slots determined by their hash codes.

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. The whole book is a collection of aphorisms like it, each nicely explained and illustrated.)

Many years ago in 6.170, 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!

### 3 Equality Properties

Let’s look first at the properties of the **equals** method. Reflexivity means that an object always equals itself; symmetry means that for any two references a and b, `a.equals(b)` must return **true** if and only if `b.equals(a)` returns **true**; transitivity means that for any three references a, b and c, if `a.equals(b)` returns **true** and `b.equals(c)` returns **true**, then `a.equals(c)` must return **true**.

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 *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:

```java
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:

```java
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:

```java
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:

```java
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 and fix this by having the `equals` method of `ColorPoint` ignore color when comparing to a non-color point:
public boolean equals (Object o) {
    if (!(o instanceof Point))
        return false;
    // if o is a normal Point, do color-blind comparison
    if (!(o instanceof ColorPoint))
        return o.equals (this);
    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 that there is no solution to this problem: the fundamental problem with inheritance is that one has multiple equals methods may be invoked when superclass and subclass objects are compared. 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 instanceof 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 and y values internally.

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

public class ColorPoint {
    private Point point;
    private Color color;

    public ColorPoint(int x, int y, Color color) {
ColorPoint no longer extends Point. In the above equals method, a ColorPoint object is never equal to a Point object. Therefore, we have symmetry and transitivity for both the equals method of Point and ColorPoint. This approach of course is called composition and we introduced it in the previous lecture.