Programming Languages and Techniques (CIS120)

Lecture 32

April 5, 2013

Equality and Hashing

When to override: Equality

Consider this example

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public int getX() { return x; }
    public int getY() { return y; }
}

// somewhere in main
List<Point> l = new LinkedList<Point>();
l.add(new Point(1,2));
System.out.println(l.contains(new Point(1,2)));
```

- What is printed to the terminal? Why?

When to override equals

- In classes that represent (immutable) values
  - String already overrides equals
  - Our Point class is a good candidate

- When there is a "logical" notion of equality
  - The collections library overrides equality for Sets
    (e.g. two sets are equal if and only if they contain the same elements)

- Whenever instances of a class can serve as elements of a set or as keys in a map
  - The collections library uses equals internally to define set membership and key lookup
  - (This is the problem with the example code.)

When *not* to override equals

- Each instance of a class is inherently unique
  - *Often* the case for mutable objects (since its state might change, the only sensible notion of equality is identity)
  - Classes that represent "active" entities and not data (e.g. threads or gui components, etc.)

- A superclass already overrides equals and provides the correct functionality.
  - Usually the case when a subclass adds only new methods, not fields

How to override equals

*See the very nicely written article "How to write an Equality Method in Java" by Oderski, Spoon, and Venners (June 1, 2009) at http://www.artima.com/lejava/articles/equality.html

The contract for equals

- The equals method implements an *equivalence relation* on non-null objects.
- It is *reflexive*:
  - for any non-null reference value x, x.equals(x) should return true
- It is *symmetric*:
  - for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true
- It is *transitive*:
  - for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is consistent:
  - for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified
- For any non-null reference x, x.equals(null) should return false.

Directly from: http://docs.oracle.com/javase/6/docs/api/java/lang/Object.html#equals(java.lang.Object)

First attempt

```
public class Point {
  private final int x;
  private final int y;
  public Point(int x, int y) {this.x = x; this.y = y;}
  public int getX() { return x; }
  public int getY() { return y; }
  public boolean equals(Point that) {
    return (this.getX() == that.getX() &&
            this.getY() == that.getY());
  }
}
```

equals was overloaded *not* overridden

```
public class Point {
  …
  // overloaded, not overridden
  public boolean equals(Point that) {
    return (this.getX() == that.getX() &&
            this.getY() == that.getY());
  }
}
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);
Object o = p2;
System.out.println(p1.equals(o));
// prints false!
System.out.println(p1.equals(p2));
// prints true!
```

The type of equals as declared in Object is:

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

The implementation above takes a Point *not* an Object!

Overriding equals, take two

Properly overridden equals

```
public class Point {
    …
    @Override
    public boolean equals(Object o) {
        // what do we do here???
    }
}
```

- Use the `@Override` annotation when you *intend* to override a method so that the compiler can warn you about accidental overloading.

- Now what? How do we know whether the o is even a Point?
  - We need a way to check the *dynamic* type of an object.

instanceof

- Java provides the `instanceof` operator that tests the *dynamic* type of any object.
  - Note: `(null instanceof C)` returns `false` for all `C`

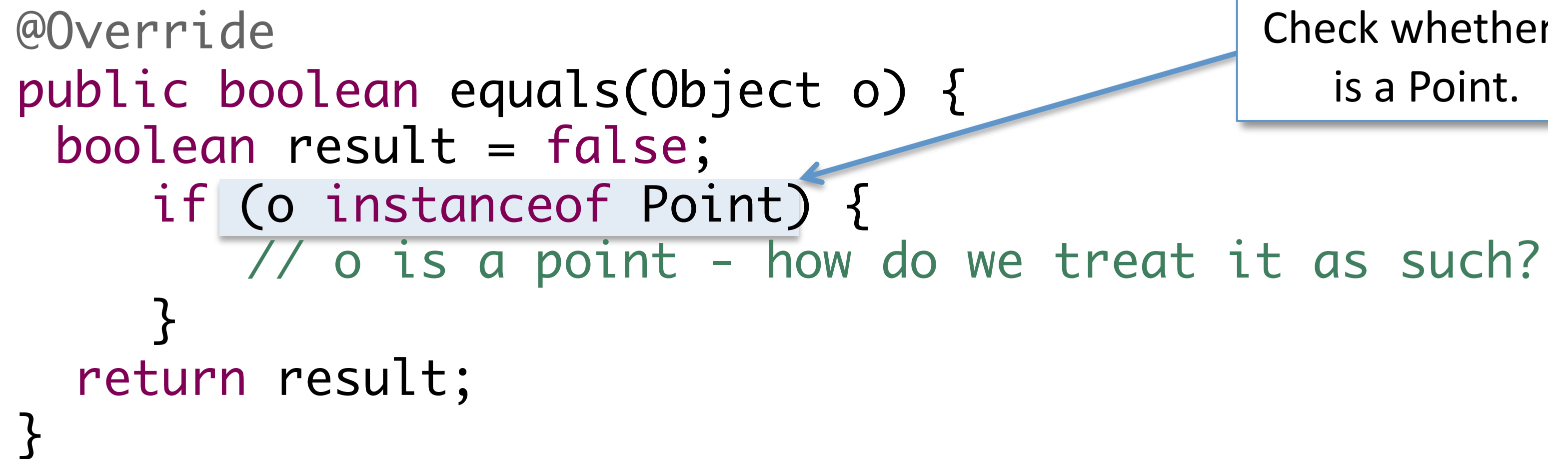
```
Point p = new Point(1,2);
Object o1 = p;
Object o2 = "hello";
System.out.println(p instanceof Point);   // prints true
System.out.println(o1 instanceof Point);  // prints true
System.out.println(o2 instanceof Point);  // prints false
System.out.println(p instanceof Object);  // prints true

// Some instanceof tests are nonsensical:
System.out.println(p instanceof String);  // compile-time error
```

- In the case of equals, instanceof is appropriate because the method behavior depends on the dynamic types of *two* objects: o1.equals(o2)
- But… use instanceof judiciously – usually, dynamic dispatch is preferred.
  - In fact, one could argue that overriding equals (and related "multimethods") is the only time one should use instanceof

Type Casts

- We can test whether o is a Point using instanceof



- Use a type *cast*: `(Point) o`
  - At compile time: the expression `(Point) o` has type `Point`.
  - At runtime: check whether the dynamic type of o is a subtype of `Point`, if so evaluate to o, otherwise raise a `ClassCastException`
  - As with instanceof, use casts judiciously – i.e. almost never. Instead use generics

Refining the equals implementation

```
@Override
public boolean equals(Object o) {
    boolean result = false;
    if (o instanceof Point) {
        Point that = (Point) o;
        result = (this.getX() == that.getX() &&
                  this.getY() == that.getY());
    }
    return result;
}
```

This cast is guaranteed to succeed.

Now the example code from the slide 2 will behave as expected.
But… are we done? Does this implementation satisfy the contract?

Equality and Subtypes

Suppose we extend Point like this

```
public class ColoredPoint extends Point {
  private final int color;
  public ColoredPoint(int x, int y, int color) {
    super(x,y);
    this.color = color;
  }
  @Override
  public boolean equals(Object o) {
    boolean result = false;
    if (o instanceof ColoredPoint) {
      ColoredPoint that = (ColoredPoint) o;
      result = (this.color == that.color &&
                super.equals(that));
    }
    return result;
  }
}
```

This version of equals is suitably modified to check the color field too.

Keyword super is used to invoke overridden methods.

Broken Symmetry

```
Point p = new Point(1,2);
ColoredPoint cp = new ColoredPoint(1,2,17);
System.out.println(p.equals(cp));
// prints true
System.out.println(cp.equals(p));
// prints false
```

- The problem arises because we mixed Points and ColoredPoints, and ColoredPoints have more data that allows for finer distinctions.
- Should a Point *ever* be equal to a ColoredPoint?

Suppose Points *can* equal ColoredPoints

```
public class ColoredPoint extends Point {
  …
  public boolean equals(Object o) {
       boolean result = false;
       if (o instanceof ColoredPoint) {
           ColoredPoint that = (ColoredPoint) o;
           result = (this.color == that.color &&
                          super.equals(that));
       } else if (o instanceof Point) {
           result = super.equals(o);
       }
       return result;
  }
}
```

- We can repair the symmetry violation by checking for Point explicitly.
- Does this work?

Broken Transitivity

```
Point p = new Point(1,2);
ColoredPoint cp1 = new ColoredPoint(1,2,17);
ColoredPoint cp2 = new ColoredPoint(1,2,42);
System.out.println(p.equals(cp1));   // prints true
System.out.println(cp1.equals(p));   // prints true(!)
System.out.println(p.equals(cp2));   // prints true
System.out.println(cp1.equals(cp2)); // prints false(!!)
```

- We fixed symmetry, but broke transitivity!
- Should a Point *ever* be equal to a ColoredPoint?

No!

A Recipe for Equality*

*Even this isn't the final story – there is another version that uses reflection to check for class names. It doesn't work well with anonymous classes or subclasses that add only methods, but is simpler in other ways. See the Odersky article for a discussion of the tradeoffs.

Add a `canEqual` method.

```
public class Point {
    …
    @Override public boolean equals(Object o) {
        boolean result = false;
        if (o instanceof Point) {
            Point that = (Point) o;
            result = (that.canEqual(this) &&
                      this.getX() == that.getX() &&
                      this.getY() == that.getY());
        }
        return result;
    }
    public boolean canEqual(Object other) {
        return (other instanceof Point);
    }
}
```

Use the canEqual method of the other object.

Expose an "instanceof" test specialized to this particular class .

Override `equals` *and* `canEqual`

```
public class ColoredPoint extends Point {
    …
    @Override
    public boolean equals(Object o) {
        boolean result = false;
        if (o instanceof ColoredPoint) {
            ColoredPoint that = (ColoredPoint) o;
            result = (that.canEqual(this) &&
                      this.color == that.color &&
                      super.equals(that));
        }
        return result;
    }
    @Override
    public boolean canEqual(Object other) {
        return (other instanceof ColoredPoint);
    }
}
```

The equals Recipe

```
public class C extends D {
  …
  @Override
  public boolean equals(Object o) {
      if (this == o) return true;
      boolean result = false;
      if (o instanceof C) {
          C that = (C) o;
          result = (that.canEqual(this) &&
                  this.field1 == that.field1 &&
                  …
                  this.field2.equals(that.field2) &&
                  super.equals(that));
          }
      return result;
  }

  public boolean canEqual(Object other) {
    return (other instanceof C);
  }
}
```

1. Override equals at the right type.

2. Return true in the case of identity. (This is an optimization.)

3. Use instanceof and cast to check the other object's type.

4. Implement the canEqual method and use the *other* object's version of it.

5. Compare all the corresponding fields, deferring to the superclass if needed.

Field Comparison

- When do you use == to compare fields?
  - for fields that store primitive types (int, boolean, etc.)
  - (often) when the field is a reference to a *mutable* object
  - when you want "shallow" equality (you don't want to follow pointers)

- When do you use equals to compare fields?
  - for references to immutable "value" types (like String or Point)
  - when you want to do "deep" equality (e.g. for singly-linked lists)
  - my model: use equals to compare objects whose representations in OCaml would not use the "ref" keyword (e.g. trees, etc.)

- Be careful about cycles!
  - It's easy to cause equals to go into an infinite loop for cyclic (often mutable) data structures.
- It's usually appropriate to defer to the superclass to check its fields.
  - But *not* in the first class to override equals! (Object uses pointer equality, remember!)
- Fields might be accessed directly or through accessor methods.

Some caveats

- Whenever you override equals you *must* also override canEqual (assuming you follow the recipe given here)
  - or provide it if the class is the first one in the inheritance tree to override equals

- Whenever you override equals you *must* also override `hashCode` in a compatible way
  - hashCode is used by the HashSet and HashMap collections