Programming Languages and Techniques (CIS120e)

Lecture 34
Dec. 3, 2010

Equality Part II, Hashing
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.
Suppose we extend Point like this

```java
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.
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?

*Even the Java standard libraries sometimes get this wrong – see the caveats/discussion about mixing Date and Timestamp objects in the JavaDocs.
Suppose Points can equal ColoredPoints

```java
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(that);
        }
        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.

```java
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.
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

```java
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 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
  – See discussion about hashing…
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 from last lecture.)
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
Hash Maps

a.k.a. Hash Tables
Hash Maps: The Big Idea

Combine:

• the advantage of arrays:
  – *efficient* random access to its elements

• with the advantage of a map datastructure
  – arbitrary keys (not just integer indices)

How?

• Create an index into an array by *hashing* the data in the key to turn it into an int
  – Java’s hashCode method maps key data to ints
  – Generally, the space of keys is much larger than the space of hashes, so, unlike array indices, hashCodes might not be unique
A schematic HashMap taking Strings (student names) to Undergraduate Majors. Here, "John Doe".hashCode() returns an integer n, its hash, such that n mod 256 is 254.
Hash Collisions

• The hashCode function should be chosen so that it is unlikely that two keys will produce the same hash.
  – However, it can happen that two keys do have the same hash value – that is, their hashes \textit{collide}

• Hash Map data structure implementations must handle such collisions to preserve the “map” semantics... there are many possible solutions.

• One simple fix is to use an array of \textit{buckets}
  – Each bucket is itself a map from keys to values (implemented by a linked list).
  – Each bucket stores the mappings for keys that have the same hash.
  – The buckets can’t use hashing to index the values – instead they use key equality (in Java, via the key’s equals method)

• To lookup a key in the Hash Map:
  – First, find the right bucket by indexing the array through the key’s hash
  – Second, search through the bucket to find the value associated with the key
Here, “Jane Smith”.hashCode() and “Joe Schmoe”.hashCode() happen to collide. The bucket at the corresponding index of the Hash Map array stores the map data.
Hash Map Performance

• Hash Maps can be used to efficiently implement Maps and Sets
  – There are many different strategies for dealing with hash collisions with various time/space tradeoffs
  – Real implementations also dynamically rescale the size of the array (which might require re-computing the bucket contents)

• If the hashCode function gives a good (close to uniform) distribution of hashes
  – the buckets are expected to be small (only one or two elements)

Whenever you override equals you must also override hashCode in a consistent way:
  – whenever o1.equals(o2) == true you must ensure that o1.hashCode() == o2.hashCode()
  – note: the converse does not have to hold:

Why? Because comparing hashes is supposed to be a quick approximation for equality.
Computing Hashes

- Java library classes come equipped with a good hashCode method
  - e.g. String
- What is a good recipe for computing hash values for your own classes?
  - intuition: “smear” the data throughout all the bits of the resulting integer

1. Start with some constant, arbitrary, non-zero int in result.
2. For each significant field f of the class (i.e. each field taken into account when computing equals), compute a “sub” hash code c for the field:
   - For boolean fields: (f ? 1 : 0)
   - For byte, char, int, short: (int) f
   - For long: (int) (f ^ (f >>> 32))
   - For references: 0 if the reference is null, otherwise use the hashCode() of the field.
3. Accumulate those subhashes into the result by doing (for each field’s c):
   result = 31 * result + c;
4. return result
Example for Point

```java
public class Point {
    ...
    @Override public int hashCode() {
        int result = 17;
        result = result * 31 + getX();
        result = result * 31 + getY();
        return result;
    }
}
```

- **Examples:**
  - `(new Point(1,2)).hashCode()` yields 16370
  - `(new Point(2,1)).hashCode()` yields 16340

- **Double check that equal points have the same `hashCode`**
  - Trivial in this case.

- **Why 17 and 31?** Primes chosen to create more uniform distributions.