Programming Languages and Techniques (CIS120)

Lecture 28
March 30, 2016

Collections and Equality
Chapter 26
Announcements

• Dr. Steve Zdancewic is guest lecturing today
  – He teaches CIS 120 in the Fall

• Midterm II is available for review
  – See Laura Fox in Levine 308

• Homework 7: PennPals
  – DUE: Tuesday, April 5th
Method Overriding
A Subclass can Override its Parent

```java
public class C {
    public void printName() { System.out.println("I’m a C"); }
}

public class D extends C {
    public void printName() { System.out.println("I’m a D"); }
}

// somewhere in main
C c = new D();
c.printName();
```

What gets printed to the console?

1. I’m a C
2. I’m a D
3. NullPointerException
4. NoSuchMethodException
A Subclass can **Override** its Parent

- Our ASM model for dynamic dispatch already explains what will happen when we run this code.
- Useful for changing the default behavior of classes.
- But... can be confusing and difficult to reason about if not used carefully.

```java
public class C {
    public void printName() {
        System.out.println("I'm a C");
    }
}

public class D extends C {
    public void printName() {
        System.out.println("I'm a D");
    }
}

// somewhere in main
C c = new D();
c.printName();
```
Overriding Example

Workspace

C c = new D();
c.printName();

Stack

Heap

Class Table

Object

String toString(){...}
boolean equals...
...

C
extends
C() {}
void printName() {...}

D
extends
D() { ... }
void printName() {...}

C c = new D();
c.printName();
c.printName();

c

D

Class Table

Object
String toString(){...}
boolean equals...
...

C
extends
C() { }
void printName() {...}

D
extends
D() { ... }
void printName() {...}
Overriding Example

Workspace

Stack

Heap

Class Table

Object
String toString() { ...
boolean equals ...
...

C
extends
C() {
void printName() { … }

D
extends
D() {
void printName() { … }

System.out.println("I’m a D");

Workspace

Stack

Heap

Class Table

Object

String toString(){...}
boolean equals...
...

C
extends
C()
{}
void printName(){...}

D
extends
D()
{
...

void printName(){...}
class C {
    public void printName() {
        System.out.println("I'm a " + getName());
    }

    public String getName() {
        return "C";
    }
}

class E extends C {
    public String getName() {
        return "E";
    }
}

// in main
C c = new E();
c.printName();

What gets printed to the console?
1. I’m a C
2. I’m a E
3. NullPointerException
Difficultly with Overriding

```java
class C {
    public void printName() {
        System.out.println("I'm a " + getName());
    }
    public String getName() {
        return "C";
    }
}
class E extends C {
    public String getName() {
        return "E";
    }
}

// in main
C c = new E();
c.printName();
```

The C class might be in another package, or a library...
Whoever wrote E might not be aware of the implications of changing `getName`.

Overriding the method causes the behavior of `printName` to change!
- Overriding can break invariants/abstractions relied upon by the superclass.
Case study: Equality
Consider this example

```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 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 gets printed to the console?

1. true
2. false

Why?

Answer: 2
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 equal elements)

• Whenever instances of a class might need to 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 \textit{not} to override equals

- When each instance of a class is inherently unique
  - \textit{Often} the case for mutable objects (since its state might change, the only sensible notion of equality is identity)
  - Classes that represent “active” entities rather than data (e.g. threads, gui components, etc.)

- When a superclass already overrides equals and provides the correct functionality.
  - Usually the case when a subclass is implemented by adding only new methods, but 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/7/docs/api/java/lang/Object.html#equals(java.lang.Object)](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object))
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());
    }
}
Gocha: *overloading*, vs. *overriding*

```java
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
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

• The `instanceof` operator tests the *dynamic* type of any object

```java
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
```

• But... use `instanceof` judiciously – usually dynamic dispatch is better.
Type Casts

• We can test whether o is a Point using instanceof

```java
@Override
public boolean equals(Object o) {
    boolean result = false;
    if (o instanceof Point) {
        // o is a point - how do we treat it as such?
    }
    return result;
}
```

Check whether o is a Point.

• 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
@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;
}
Equality and Subtypes
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.

Keyword `super` is used to invoke overridden methods.
Broken Symmetry

The problem arises because we mixed Points and ColoredPoints, but ColoredPoints have more data that allows for finer distinctions.

Should a Point ever be equal to a ColoredPoint?

```java
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
```
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(o);
        }
        return result;
    }
}
```

I.e., we repair the symmetry violation by checking for Point explicitly

Does this really work? (1=yes, 2=no)
Broken Transitivity

- We fixed symmetry, but broke transitivity!
- Should a Point ever be equal to a ColoredPoint? No!
Should equality use `instanceof`?

- To correctly account for subtyping, we need the classes of the two objects to match *exactly*.
- `instanceof` only lets us ask about the subtype relation
- How do we access the dynamic class?

```java
class C { }
void printName() {}

class D extends C { }

Object o = new String();

The `o.getClass()` method returns an object that represents `o`'s dynamic class.

Reference equality `==` on class values correctly checks for class equality (i.e. there is only ever *one* object that represents each class).
Correct Implementation: Point

```java
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Point other = (Point) obj;
    if (x != other.x)
        return false;
    if (y != other.y)
        return false;
    return true;
}
```

Check whether `obj` is a `Point`. 
Equality and Hashing

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

• Forgetting to do this can lead to extremely puzzling bugs!
Overriding Equality in Practice

• Eclipse can autogenerate equality methods of the kind we developed.
  – But you need to specify which fields should be taken into account.
  – and you should know why some comparisons use == and some use .equals