Static Overloading, and Equality

In lecture 13, we looked at the Java type system more carefully, especially the effect of subtyping on the *static type* of data — and its consequence of loss of static type information that can lead Java to give type errors and the use of parametric polymorphism to avoid such loss of information — and also the use of dynamic dispatch for method invocation, that says that the method invoked on an object is the one given by its *dynamic type*, that is, the type is has at run time.

We now have enough information to start revisiting equality, and see why the implementation I gave you a couple of lectures back is in fact not good at all. There is a small problem with it, and a big problem with it. And the most obvious correction is also wrong. That turns out to be a big problem — getting equality right in Java, or in fact in any object-oriented language, is a pretty big issue. Various approaches have been proposed, none completely satisfactory.

Recall that I gave you the following implementation for equality. Consider the following implementation of the Point ADT from Lecture 7, augmented with an `equals` operation and a `hashCode` operation (not the best, but it will do) — because we always need to redefine `hashCode` when redefining `equals`:

```
public class Point {
  private int xpos;
  private int ypos;

  // needed - Java specific incantation
  protected Point () { }

  private Point (int x, int y) {
    xpos = x;
    ypos = y;
  }

  public static Point create (int x, int y) {
    return new Point(x,y);
  }

  public int xPos () { return this.xpos; }
```

```
  public int yPos () { return this.ypos; }

  public Point move (int dx, int dy) {
    return create(this.xpos + dx, this.ypos + dy);
  }

  public boolean equals (Point p) {
    return (p.xPos()==this.xPos() && p.yPos()==this.yPos());
  }

  public boolean equals (Object obj) {
    return false;
  }

  public int hashCode () {
    return this.xPos()+this.yPos();
  }
}
```

The code for `equals` is very pretty. It's also wrong. To understand what goes on, we need to understand how Java deals with the fact that there are two methods of the same name (but with different argument types). This is called *overloading* — multiple methods, same name, different types for the arguments. Clearly, which method gets called depends on the type of the argument. But which type? It turns out that which methods gets called depends on the *static type of the argument*, and not its dynamic type. The following code illustrates what is going on:

```
public static class Foo {
  public Foo () {}

  public void print (Foo f) {
    System.out.println("Printing a foo");
  }

  public void print (Object obj) {
    System.out.println("Printing an object");
  }
}
```

If we execute:

```
Foo f = new Foo();
```

```
f.print(f);
```

we get the output:

```
Printing a foo
```

where if we execute:

```
Object fobj = new Foo();
f.print(fobj);
```

we get the output:

```
Printing an object
```

So different output, even though at run time, of course, the values `f` and `fobj` have the same dynamic type `Foo`. But `f` has static type `Foo` while `fobj` has static type `Object` — the assignment loses static type information, and this has an effect on which method gets called. The above behavior is sometimes called *static overloading* — overloading of methods resolved by the static type of arguments.[1]

How does this affect equality? Well, the fact that I used static overloading to implement equality means that we can get "wrong" answers to equality test if we lose static information. For instance, using the definition of `Point` above, I could write:

```
Point p1 = Point.create(0,0);
Object p2 = Point.create(0,0);
if (p1.equals(p2))
  System.out.println("The points are equal");
else
  System.out.println("The points are not equal");
```

and executing this code gives back:

```
The points are not equal
```

Now, we could argue whether or not that's the behavior we want. I think it is not. `p1` and `p2` are really two points, and they are really equal, and if we cast `p2` to a `Point` then `p1` and `p2` compare as equal, and a cast should really not affect run-time behavior. (That's probably the right invariant to keep in mind: casts do not affect run-time behavior, short of

[1]Not every language with overloading uses static overloading. The alternative is of course *dynamic overloading*, where the method that gets called depends on the dynamic type of arguments.

sometimes throwing exceptions.) So if you buy this, then implementing equality with static overloading is just wrong. That's the general consensus.

(There is another big problem with the above implementation of equality, one that I actually missed the first time around, and that's that it does not deal well with `null`. Recall that `null` is a value that can have any object type, but when you try to invoke a method on `null` you get an exception. There are no checks for `null` above, meaning that it's entirely possible to write:

```
Point p1 = Point.create(0,0);
Point p2 = null;
if (p1.equals(p2))
  System.out.println("The points are equal");
else
  System.out.println("The points are not equal");
```

and the code explodes when you execute it. It's easy to fix by implementing the first `equals` method as:

```
public boolean equals (Point p) {
  if (p==null)
    return false;
  return (p.xPos()==this.xPos() && p.yPos()==this.yPos());
}
```

but it's already much less pretty, even before accounting for the fact that it does not work if you buy my argument above.)

So if we do not use static overloading, how do we implement equality? Well, we want only one equality method, and it has to be able to taking anything as an argument. So this means having a single method `equals` taking in an `Object` argument, which is the class of which every class in Java is a subclass. Then, depending on the *dynamic type* of the argument, if that type is `Point`, we can check the `xPos()` and `yPos()` positions, otherwise, if the dynamic type of the argument is not `Point`, we return false.

But how do we check that the dynamic type of the argument is `Point`? Well, to a first approximation, we can think of using `instanceof`, that I mentioned in Lecture 13. Recall that (*exp* `instanceof` *cls*) returns true exactly when the result of evaluating *exp* is an instance of (i.e., an object that is an instance of a subclass of) class *cls*. This leads to the following implementation of `equals`, which is similar to what you saw in 213:

```
public boolean equals (Object obj) {
  if (obj instanceof Point) {
    Point p = (Point) obj;
```

```
      return (p.xPos()==this.xPos() && p.yPos()==this.yPos());
    }
    return false;
  }
```

And it gives us at least the behavior we originally wanted — executing

```
  Point p1 = Point.create(0,0);
  Object p2 = Point.create(0,0);
  if (p1.equals(p2))
    System.out.println("The points are equal");
  else
    System.out.println("The points are not equal");
```

gives us

```
  The points are equal
```

Great. And note that we take care of `null` automatically, since it turns out that (`null instanceof` *cls*) is false for any class *cls*, so that:

```
  Point p1 = Point.create(0,0);
  Point p2 = null;
  if (p1.equals(p2))
    System.out.println("The points are equal");
  else
    System.out.println("The points are not equal");
```

gives back

```
  The points are not equal
```

as expected.

Great. Not as pretty as using overloading, and there's that annoying `instanceof` in there that is a big wart, but it is serviceable code.

And it's very much wrong, although for a slightly more subtle reason. Let me be precise: by itself, the code is not wrong, but if we define a subclass of `Point`, then `equals` ceases to work as we want it to work — in particular, it fails the implicit specification of it being symmetrical. Consider the implementation of `Point` with the last definition of `equals` in terms of `instanceof`, and consider the following subclass of `CPoint`, with a similar definition of `equals`:

```
public class CPoint extends Point {
  private int xpos;
  private int ypos;
  private String color;

  // needed - Java specific incantation
  protected CPoint () { }

  private CPoint (int x, int y, String c) {
    xpos = x;
    ypos = y;
    color = c;
  }

  public static CPoint create (int x, int y, String c) {
    return new CPoint(x,y,c);
  }

  public int xPos () { return this.xpos; }

  public int yPos () { return this.ypos; }

  public String color () { return this.color; }

  public CPoint move (int dx, int dy) {
    return create(this.xpos + dx, this.ypos + dy, this.color);
  }

  public boolean equals (Object obj) {
    if (obj instanceof CPoint) {
      CPoint p = (CPoint) obj;
      return (p.xPos()==this.xPos() && p.yPos()==this.yPos()
              && p.color().equals(this.color()));
    }
    return false;
  }

  public int hashCode () {
    return this.xPos()+this.yPos()+this.color().hashCode();
  }
}
```

Now consider the following piece of code:

```
Point p1 = Point.create(0,0);
CPoint p2 = CPoint.create(0,0,"red");

if (p1.equals(p2))
  System.out.println("The points are equal");
else
  System.out.println("The points are not equal");

if (p2.equals(p1))
  System.out.println("The points are equal");
else
  System.out.println("The points are not equal");
```

This should return the same result twice, because symmetry says that `p1.equals(p2)` should be the same as `p2.equals(p1)`. But it's not:

```
The points are equal
The points are not equal
```

That's because `CPoint` is a subclass of `Point` (so that an object with dynamic type `CPoint` returns true for `instanceof Point`), but `Point` is not a subclass of `CPoint` (so that an object with dynamic type `Point` returns false for `instanceof CPoint`).

There aren't a great many ways of solving this particular quandary. Subclassing, dynamic dispatch, and symmetry for equality can be seen as incompatible. So people generally accept the implementation of `equals` above in terms of `instanceof` and live with the non-symmetry.

Sometimes, we can do a bit better. Continuing with the point/colored point example: a colored point is not a point, meaning that if we try to compare a point against a colored point, we should get false for their equality every time. After all, colored point have a color. We get that in one direction, but not the other: if `p1` is a point and `p2` is a colored point, then `p1.equals(p2)` is true if the `xPos()` and `yPos()` of the points are equal. One way to ensure that points are only compared with points is to ask not only that the argument of `equals` be an `instanceof Point` (or `CPoint`, or whatever), but that it *exactly is a* `Point` (or `CPoint` or whatever), and not an instance of a subclass of `Point`.

There is one way in Java to do that, and that's using *reflection*, which gives you run-time access to information that is otherwise not available, like the dynamic type of a value. Here is the updated code for `equals` in `Point`:

```
public boolean equals (Object obj) {
  if (obj!=null && obj.getClass().equals(this.getClass())) {
```

```
    Point p = (Point) obj;
    return (p.xPos()==this.xPos() && p.yPos()==this.yPos());
  }
  return false;
}
```

Note that we check explicitly for `null` (returning `false` if it happens to be), and that we check that the actual class (i.e., the dynamic type) of `obj` is the same as the actual class of `this`. So `Point` is equal to `Point`, but `CPoint` is not equal to `Point`, so we get the behavior we want, assuming we change `CPoint`'s `equals` method to do the same kind of check:

```
Point p1 = Point.create(0,0);
CPoint p2 = CPoint.create(0,0,"red");

if (p1.equals(p2))
  System.out.println("The points are equal");
else
  System.out.println("The points are not equal");

if (p2.equals(p1))
  System.out.println("The points are equal");
else
  System.out.println("The points are not equal");
```

yields:

```
The points are not equal
The points are not equal
```

What's the rub? Well, this is fine for `Point`/`CPoint`, but we can imagine deriving a subclass of `Point` called something like `AlternatePoint` that has exactly the same implementation as `Point` — don't ask why we want to do that, someone for some reason did that and you have to deal with it — well, a `Point` and `AlternatePoint` should be comparable for equality, by just comparing the `xPos()` and `yPos()`, but writing an equality method that is symmetrical in that context is near impossible, unless you bake in the possible subclasses in the implementation of `Point`, and that's completely unmaintainable (not to mention that it doesn't scale.)

Bottom line: equality in Java, not easy.