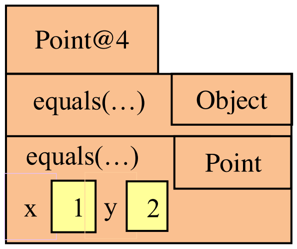
Function equals

Superest class Object has a function equals. Let b and c be (pointers to) objects of class Object. Then,

b.equals(c) is true if and only if b == c is true —b and c point to the same object.

For some classes, it makes sense to override function equals. Class String overrides function equals: for String objects s1 and s2, s1.equals(s2) is true exactly when s1 and s2 contain the same string of characters.

It also makes sense to override equals in a class Point whose objects represent points (x, y) in the plane. Here's a good specificiation of equals in Point:



```
/** Return true iff this and ob are objects of the same
  * class and they have the same x, y values. */
@Override public boolean equals(Object ob)
```

(Note that the type of parameter ob must be Object; if you change it to Point, this function does not override function equals in class Object.)

But not all classes need to override equals. Class Throwable and all of its subclasses don't. For example, the value of this expression is false, even though the detail message in both objects is null.

```
new ArithmeticException().equals(new ArithmeticException())
```

Below, we discuss implementing function equals and a pitfall to avoid.

Properties of equality

The specification of equals in class Object says that any function that overrides equals should satisfy the traditional properties of equality:

Reflexive: x = x. In Java: b.equals(b) should be true.

Symmetric: x = y if and only if y = x. In Java: for non-null b and c, b.equals(c) and c.equals(b) are both true or both false.

Transitive: if x = y and y = z, then x = z. In Java: if b.equals(c) is true and c.equals(d) is true, then b.equals(d) is true.

And in Java: for non-null b, b.equals(null) is false.

It's our job when overriding function equals to make sure these properties hold.

Writing equals in class Point

Look in the box below at the spec of function equals in class Point. There are two simple ways to test the class of parameter ob: use operator instanceof and use function getClass[1]. Because of the way equals is specified, getClass has to be used —we comment later on the use of instanceof. Remember, this function goes in class Point.

```
/** Return true iff this and ob are objects of the
  * same class and they have the same x, y values */
@Override public boolean equals(Object ob) {
    if (ob == null  ||  getClass() != ob.getClass())
        return false;
    Point op= (Point) ob;
    return x == op.x  &&  y == op.y;
}
```

There are three parts to the body.

(1) The if-statement returns false if ob is null or this object and ob are not of the same class. The test on ob null is needed so that ob.getClass() can be called.

(2) If they are of the same class, ob is cast down to class Point, so that its fields can be accessed, and stored in a local variable.

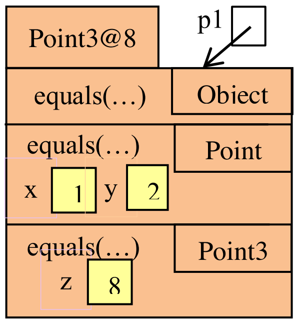
(3) The return statement returns true iff fields x and y in the two objects are equal.

Usually, all three parts are needed: testing for the right class, casting down, and checking field equality.

[1] If you are not familiar with instanceof and getClass, please study the JavaHyperText entry "instanceof".

Function equals

Writing a subclass of class Point



Suppose we want to implement a class that maintains a point (x, y, z) in 3-dimensional space. Rather than write a completely new class, we extend class Point. This may not be the best strategy for implementing 3-dimensional points, but it helps us illustrate how to deal with subclasses. Object p1 of Point3 appears to the right.

Below, we write function equals in Point3 to override equals in Point:

```
/** Return true iff this and ob are objects of the
  * same class and have the same x, y, and z values */
@Override public boolean equals(Object ob) {
      if (ob == null  ||  getClass() != ob.getClass()) return false;
      if (!super.equals(ob)) return false;
      return z == ((Point3) ob).z;
}
```

Read the specification carefully, and note how the first if-statement returns false if this and ob are not of the same class —again the test on ob null is needed to protect the call ob.getClass(). Second, the call on super.equals checks that fields x and y of the two objects are the same. This is practically the only way to check this property since the superclass fields are private. Third, we have the necessary cast to Point3 so that the equality of the two fields z can be tested.

Whenever you are writing a function equals in a subclass that has fields, it should probably look like this one. This method equals illustrates the systematic way to test all that has to be tested.

Note that if equals in class Point is called, it doesn't matter whether the two objects in question are of class Point or of class Point3; the two objects just have to have the same type.

Why the use of instanceof leads to problems

If class Point was never subclassed, it would be alright to use operator instanceof instead of function getClass. But if Point has a subclass, using instanceof may lead to problems. We illustrate.

To the right, we rewrite the specs of method equals in classes Point and Point3. Because of the specs, we can use instanceof in the method body. Study the methods. No test on ob null is needed because   null instanceof Point   is false.

Put this method in class Point

```
/** Return true iff ob is a Point and has the
  *  same x and y values as this object. */
@Override public boolean equals(Object ob) {
      if (!(ob instanceof Point)) return false;
      Point op= (Point) ob;
      return x == op.x  &&  y == op.y;
}
```

Put this method in class Point3

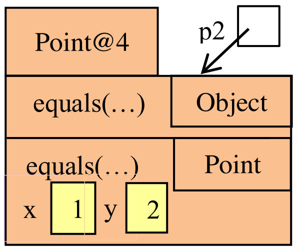
```
/** Return true iff ob is a Point3 and has the
  * same x, y, and z values as this. */
@Override public boolean equals(Object ob) {
     if (!(ob instanceof Point3)) return false;
     if (!super.equals(ob)) return false;
     Point3 op= (Point3) ob;
     return z == op.z;
}
```

To the right at the top of the page is variable p1 pointing at a Point3 object. To the right below is variable p2 pointing at a Point object.

The call p2.equals(p1) returns true, since p1 has a partition named Point and the x and y fields of p1 and p2 contain the same values. Note that only equals in Point is called.

The call p1.equals(p2) returns false, since p2 does not have a partition named Point3.

Thus, the symmetric property of function equals doesn't hold, and the implementation is faulty.



Discussion

If a class C is not going to be subclassed, for example if it is declared as final, it doesn't matter whether getClass or instanceof is used to test the class of parameter ob. But if C will be subclassed, getClass should probably be used. We say "probably" because it all depends on what the writer of class C means by equality. Any interpretation of equality will do as long as it is reflexive, symmetric, and transitive.