Effective Java, Chapter 3: Methods Common to All Objects

Last Updated Fall 2012

Paul Ammann

Agenda

- Material From Joshua Bloch
  - Effective Java: Programming Language Guide
- Cover Items 8 through 12
  - Methods Common to All Objects
- Moral:
  - Contract model = Industry Practice!

Item 8

- Obey the general contract when overriding `equals()`
- Overriding seems simple, but there are many ways to get it wrong.
- Best approach – Avoid! Works if:
  - Each instance of a class is unique
  - You don't care if class has logical equality
  - The superclass equals is satisfactory
  - Class is not public and equals never used

General contract for equals

- Reflexive
  - `x.equals(x)` must be true
- Symmetric
  - `x.equals(y)` iff `y.equals(x)`
- Transitive
  - If `x.equals(y) && y.equals(z)`
  - Then `x.equals(z)`
- Consistency…
- Null values:
  - `x.equals(null)` is always false

How hard could this be?

- Reflexivity is pretty much automatic
- Symmetry is not:
  - Example `CaseInsensitiveString`

```
private String s;
// Broken - violates symmetry
@Override public boolean equals (Object o) {
   if (o instanceof CaseInsensitiveString)
      return s.equalsIgnoreCase(
         ((CaseInsensitiveString) o).s);
   if (o instance of String) // Not Symmetric!
      return s.equalsIgnoreCase((String) o);
   return false;
}
```

Why does this violate symmetry?

- Consider this code:

```
Object x = new CaseInsenstiveString (“abc”);
Object y = “Abc”;  // y is a String
if (x.equals(y))  {…}  // evaluates true, so execute
if (y.equals(x))  {…}  // evaluates false, so don’t…
```

- Dispatching of `equals()` calls
  - First `equals()` call to `CaseInsensitiveString`
  - Second `equals()` call to `String`
- This is horrible!

Correct Implementation

- Avoid temptation to be "compatible" with the `String` class:

```
// CaseInsensitiveString is not a subclass of String!
private String s;
@Override public boolean equals (Object o) {
   return (o instanceof CaseInsensitiveString)
      &&
      ((CaseInsensitiveString) o).s.
      equalsIgnoreCase(s);
}
```

Symmetry and Transitivity

- Surprisingly difficult – general result about inheritance
- Example:
  - A 2D `Point` class
    - State is two integer values x and y
    - `equals()` simply compares x and y values
  - An extension to include color
    - public class `ColorPoint` extends `Point`
    - What should `equals()` do?

Preliminaries: What does equals in Point look like?

```
public class Point {  // routine code
   private int x; private int y;
   ...
   @Override public boolean equals(Object o) {
      if (!(o instanceof Point))
         return false;
      Point p = (Point) o;
      return p.x == x && p.y == y;
   }
}
```

Choice 1 for equals() in ColorPoint

- Have equals() return true iff the other point is also a ColorPoint:

```
// Broken - violates symmetry
 @Override public boolean equals(Object o) {
    if (!(o instanceof ColorPoint))
       return false;
    ColorPoint cp = (ColorPoint o);
    return super.equals(o) &&
        cp.color == color;
 }
```

Problem

- Symmetry is broken
- Different results if comparing:

```
ColorPoint cp = new ColorPoint (1, 2, RED);
Point p = new Point (1,2);
// p.equals(cp), cp.equals(p) differ
```

- Unfortunately, `equals()` in `Point` doesn't know about `ColorPoints`
  - Nor should it…
- So, try a different approach…

Choice 2 for equals() in ColorPoint

- Have `equals()` ignore color when doing "mixed" comparisons:

```
// Broken - violates transitivity
@Override public boolean equals(Object o) {
    if (!(o instance of Point)) return false;
    // If o is a normal Point, be colorblind
    if (!o instanceof ColorPoint)
       return o.equals(this);
    ColorPoint cp = (ColorPoint o);
    return super.equals(o) && cp.color == color;
 }
```

Now symmetric, but not transitive!

- Consider the following example

```
ColorPoint p1 = new ColorPoint(1,2,RED);
Point p2 = new Point(1,2);
ColorPoint p3 = new ColorPoint(1,2,BLUE);
```

- The following are true:
  - `p1.equals(p2)`
  - `p2.equals(p3)`
- But not `p1.equals(p3)`!

The real lesson

- There is no way to extend an *instantiable* class and add an aspect while preserving the equals contract.
  - Note that abstract superclass definitions of `equals()` are fine. (See Bloch Item 20)
- Wow! Inheritance is hard!
- Solution: Favor composition over inheritance (Item 16).
- Note: This was not well understood when some Java libraries were built…

How to implement equals()

- Use == to see if argument is a reference to this (optimization)
- Use instanceof to check if argument is of the correct type (properly handles null)
- Cast the argument to the correct type
- Check each “significant” field
- Check reflexivity, symmetry, transitivity

Be sure to maintain Liskov Substitution Principle

- Rumor has it you can use `getClass()` instead of `instanceof`
  - Bloch argues that this is simply wrong
  - See Wagner, Effective C#, Item 9, for an alternate viewpoint

```
// Broken - violates Liskov substitution principle
@Override public boolean equals(Object o) {
    if (o == null || o.getClass() != getClass)
        return false;
    Point p = (Point o);
    return p.x == x && p.y == y;
}
```

Client Use of Point Class

- ```java
  // Initialize UnitCircle to contain Points on unit circle
  private static final Set<Point> unitCircle;
  static {
     unitCircle = new HashSet<Point>();
     unitCircle.add(new Point( 1,  0));
     unitCircle.add(new Point( 0,  1));
     unitCircle.add(new Point(-1,  0));
     unitCircle.add(new Point( 0, -1));
  }
  public static boolean onUnitCircle (Point p) {
     return unitCircle.contains(p);
  }
  ```

Question: Which Point objects should onUnitCircle() handle?
  ```

Completion of prior example

- Now consider a different subclass `CounterPoint`
  - Question: What happens to clients of `Point`?
  - Answer: `CounterPoint` objects behave badly ☹

```
public class CounterPoint extends Point
    private static final AtomicInteger counter =
        new AtomicInteger();

    public CounterPoint(int x, int y) {
        super (x, y);
        counter.incrementAndGet();
    }
    public int numberCreated() { return counter.get(); }
}
```

What not to do

- Don’t be too clever
- Don’t use unreliable resources, such as IP addresses
- Don’t substitute another type for Object
  - `@Override public boolean equals (MyClass o)`
    - Wrong, but `@Override` tag guarantees compiler will catch problem
  - Overloads `equals()` – does not override it!
- Don’t throw `NullPointerException` or `ClassCastException`

Item 9

- Always override `hashCode()` when you override `equals()`
- Contract:
  - `hashCode()` must return same integer on different calls, as long as `equals()` unchanged
  - If `x.equals(y)`, then x, y have same hashcode
  - It is **not** required that unequal objects have different hashcodes.

Second provision is key

- Suppose `x.equals(y)`, but x and y have different values for `hashCode()`
- Consider this code:

```
Map m = new HashMap();
m.put(x, "Hello");  // expect x to map to Hello
// m.get(y) should return Hello,
// since x.equals(y), but it doesn't!
```

- Ouch!

How to implement hashCode

- Avoid really bad implementations
  - `@Override public int hashCode() { return 42;}`
  - Hash table now performs terribly (but, at least, correctly…)
- Start with some nonzero value (eg 17)
- (Repeatedly) compute int hashCode "c" for each "significant field"
  - Various rules for each data type
- Combine: `result = result*37 + c;`

Optimize hashCode() for immutable objects

- No reason to recompute hashcode
- Maybe no reason to compute at all!

```
// Lazy initialization example
 private int hashCode = 0;
 @Override public int hashCode() {
     if (hashCode == 0)
         { …}  // needed now, so compute hashCode
     else return hashCode;
 }
```

Item 10

- Always override `toString()`
- Return all the "interesting" information in an object
  - `toString()` simply implements the Abstraction Function for an object
  - `toString()` values **must not change** if representation changes
- Document intentions with respect to format
  - Clients may (unwisely) decide to depend on format
  - Provide getters for values `toString()` provides
    - Do **not** force clients to parse String representation

Item 11

- Override `clone()` judiciously
- Cloneable is a “mixin” interface
  - Unfortunately, it fails to provide any methods
    - `clone()` is defined in `Object` (protected)
- Contract:
  - Create a copy such that `x.clone() != x`
  - `x.clone().getClass() == x.getClass()`
  - Should have `x.clone().equals(x)`
  - No constructors are called

What a strange contract

- The requirement on classing is too weak
  - A programmer calling `super.clone()` wants an instance of the subclass, not the superclass.
  - The only way to do this is to call `super.clone()` all the way up to `Object`.
  - Explicit use of constructors gives the wrong class.
- Rule: Always implement `clone()` by calling `super.clone()`.

The role of mutability

- If a class has only primitive fields or immutable references as fields, `super.clone()` returns exactly what you want
- For objects with mutable references, “deep copies” are required.
- Example: cloning a `Stack` class that uses a `Vector` for a representation.
  - Representation `Vector` must also be cloned.
  - So, call `super.clone()`, then clone `Vector`

Other Cloning problems

- Cloning may be a problem with final fields
- Cloning recursively may not be sufficient
- Result:
  - You may be better off not implementing Cloneable
  - Providing a separate copy mechanism may be preferable.
    - Copy Constructor: `public Yum (Yum yum)`
    - Factory: `public static Yum newInstance(Yum yum)`

Item 12

- Consider Implementing Comparable
- Contract
  - Returns negative, zero, or positive depending on order of this and specified object
  - `sgn(x.compareTo(y) == -sgn(y.compareTo(x))`
  - `compareTo()` must be transitive
  - If `x.compareTo(y) == 0`, x and y must consistently compare to all values z.
  - Recommended that `x.compareTo(y) == 0` iff `x.equals(y)`
  - Note that `compareTo()` can throw exceptions

Elements of the contract

- The same issue with `equals()` arises in the case of inheritance:
  - There is simply no way to extend an instantiable class with a new aspect while preserving the `compareTo()` contract.
  - Same workaround – Favor composition over inheritance
- Some Java classes violate the consistency requirement with `equals()`.
  - Example: The `BigDecimal` class

BigDecimal Example

```
//This is horrible!
Object x = new BigDecimal("1.0");
Object y = new BigDecimal("1.00");
// !x.equals(y), but x.compareTo(y) == 0
Set s = new HashSet(); Set t = new TreeSet();
s.add(x); s.add(y);
// HashSet uses equals, so s has 2 elements
t.add(x); t.add(y);
// TreeSet uses compareTo, so t has 1 element
```