Mutation vs. Immutables

Spring 2012
To be (mutated) or not to be (Mutated) that is the question
pop quiz

what happens when this code is executed?

```java
String s = "hello";
s.concat(" world");
System.out.println (s);
s = s.concat(" world");
System.out.println (s);
```

hello

hello world

and how about this?

```java
StringBuffer sb = new StringBuffer ("hello");
sb.append(" world");
System.out.println (sb);
StringBuffer sb2 = sb;
StringBuffer sb2 = sb;
sb2.append ("!");
System.out.println (sb);
```

hello world

hello world!
mutable vs. immutable

**String is an immutable datatype**
- computation creates new objects with producers

```java
class String {
    String concat (String s);
    ...
}
```

**StringBuffer is a mutable datatype**
- computation gives new values to existing objects with mutators

```java
class StringBuffer {
    void append (String s);
    ...
}
```
solutions

what you needed to know to answer correctly

immutable and mutable types

- String is immutable, StringBuffer is mutable
- method call on immutable object can’t affect it

assignment semantics

- the statement \( x = e \) makes \( x \) point to the object that \( e \) evaluates to

aliasing

- the statement \( x = y \) makes \( x \) point to the same object as \( y \)
- subsequent mutations of the object are seen equivalently through \( x \) and \( y \)
- since immutable objects can’t be mutated, sharing is not observable
how mutation happens

through field setting

- statement \( x.f = y \) makes \( f \) field of \( x \) point to object \( y \)

through array update

- statement \( a[i] = y \) makes \( \text{element}_i \) ‘field’ of \( a \) point to object \( y \)
heap reachability

an assignment or field set can leave an object unreachable from example before

- after these statements
  ```java
  String s = "hello";
  s = s.concat(" world");
  ```
  the two string literal objects are unreachable

once an object is unreachable

- it cannot be reached again
- so removing it will not be observable

garbage collector (aka “automatic memory management”)

- marks unreachable objects, then deallocates them
heap reachability

an assignment or field set can leave an object unreachable

from example before

- after these statements
  ```java
  String s = "hello";
  s = s.concat(" world");
  ```
- the two string literal objects are unreachable

once an object is unreachable

- it cannot be reached again
- so removing it will not be observable

garbage collector (aka “automatic memory management”)

- marks unreachable objects, then deallocates them
conceptual leaks

storage leak
➢ use of memory grows, but active state isn’t growing

no storage leaks in garbage-collected language?
➢ unfortunately, can still happen

What is wrong with this code?

```java
public class ArraySet {
    private Object[] elements;
    private int size;

    ... 

    public void delete (Object o) {
        for (int i = 0; i < size; i++) {
            if (elements[i].equals(o)) {
                elements[i] = elements[size-1];
                elements[size-1] = null;
                size--;
            }
        }
    }
}
```
EQUALITY
fundamentals

objects often used as keys

- need to compare them

- eg, `Literal` used as key in `Environment`

Java convention

- the class `Object` has a method that every class inherits
  - `Object.equals: Object -> boolean`

- by convention, this method is used to compare objects for equality
- collections especially assume this: call `equals` on keys
- the inherited method is usually wrong for immutable types
  so must override by explicitly declaring a method
  - `MyType.equals: Object -> boolean`
why inherited equality fails

the problem

- `Object.equals` compares objects with `==`
- this makes any two distinct objects unequal
- even if they have the same value

example

- the “same” pairs are unequal:

```java
public class Pair {
    private final int fst, snd;
    public Pair (int f, int s) {fst=f; snd=s;}

    public static void main (String[] args) {
        Pair p1 = new Pair (1, 2);
        Pair p2 = new Pair (1, 2);
        System.out.println (p1 == p2 ? "yes" : "no");
        System.out.println (p1.equals(p2) ? "yes" : "no");
    }
}
```
standard equals method

correct code for Pair.equals

- compare the fields

  ```java
  @Override
  public boolean equals (Object that) {
    if (this == that) return true;
    if (!(that instanceof Pair)) return false;
    Pair p = (Pair) that;
    return p.fst == fst && p.snd == snd;
  }
  ```

remember: comparison is with any object reference

- need to check type of arg, and whether null
- you may be tempted to write this, but don’t: it will just overload equals

  ```java
  public boolean equals (Pair that) {...}
  ```

- write @Override and compiler will catch the bug
a running example

a duration class

- represents durations measured in minutes

```java
public class Duration {
    private final int hours;
    private final int mins;
    public Duration(int h, int m) {hours = h; mins = m;}
    public int getMins() {return hours*60 + mins;}
}
```
abstraction function

```java
Duration d1 = new Duration(1, 2);
Duration d2 = new Duration(1, 3);
Duration d3 = new Duration(0, 62);
```
Bug #1: Transitivity

```java
public class Duration {
    private final int hours;
    private final int mins;
    static final int CLOCK_SKEW = ...;
    public boolean equals(Duration d) {
        if (d == null) return false;
        return Math.abs(d.getMins() - this.getMins()) < CLOCK_SKEW;
    }
}
```

![Diagram showing a counterexample to transitivity]
bug #2:

```java
public class Duration {
    private final int hours;
    private final int mins;
    public Duration(int h, int m) {
        hours = h;
        mins = m;
    }
    public boolean equals(Duration d) {
        return d.getMins() == this.getMins();
    }
}
```

What is the bug here?

```
Duration d1 = new Duration(1,2);
Duration d2 = new Duration(1,2);
System.out.println(d1.equals(d2));  // Prints true

Object d1 = new Duration(1,2);
Object d2 = new Duration(1,2);
System.out.println(d1.equals(d2));  // Prints false!!
```
explaining bug #2

what’s going on?
• we’ve failed to **override** Object.equals
  ➢ method is chosen using compile-time type
  ➢ method has been overloaded, not overridden

```java
class Object {
    public boolean equals (Object o) { return o == this; }
}
class Duration extends Object {
    public boolean equals (Object o) { return o == this; }
    public boolean equals (Duration d) {
        return d.getMins() == this.getMins();
    }
}
```
equality and subclassing

```java
public class ShortDuration extends Duration {
    private final int secs;
    ...
    private ShortDuration (int h, int m, int s) {...};
    public int getSecs () {return 3600*hours + 60*mins + secs;}
    ...
}
```

how should equality be determined?

- can’t rely on inherited equals method, because seconds are ignored
bug #3

@Override
public boolean equals(Object o) {
    if (! (o instanceof ShortDuration))
        return false;
    ShortDuration d = (ShortDuration) o;
    return d.getSecs () == this.getSecs();
}

will this work?
• NO, now it’s not symmetric!

Duration d1 = new ShortDuration(1,2,3);
Duration d2 = new Duration(1,2);
System.out.println(d1.equals(d2));
System.out.println(d2.equals(d1));
yet another attempt; this time not transitive

duration d1 = new ShortDuration(1,2,3);
duration d2 = new Duration(1,2);
duration d3 = new ShortDuration(1,2,4);
System.out.println(d1.equals(d2));
System.out.println(d2.equals(d3));
System.out.println(d1.equals(d3));
solving the subclassing snag

no really satisfactory solution

superclass equality rejects subclass objects
  • can write this
    • if (!o.getClass().equals(getClass())) return false;
  • but this is inflexible: can’t extend just to add functionality, eg

better solution
  • avoid inheritance, and use composition instead
  • see Bloch, Effective Java, Item 14

```java
public class ShortDuration {
    private final Duration dur;  //
    private final int secs;
    ...
    private ShortDuration (int h, int m, int s) {...};
    public int getSecs () {return 3600*dur.hours + 60*dur.mins + secs;}   ... }
```
Interning

If two objects start equal, they will always be equal

The following are impossible to tell apart:

- Two references point to the same object
- Two references point to identical objects

That wouldn’t be true if the objects were mutable!
a design puzzle

public class C {
    private String s;
    public static Map<C, C> allocated = new ListMap<C, C>();
    public C intern () {
        C c = allocated.get(this);
        if (c == null) {
            allocated = allocated.put(this, this);
            return this;
        }
        return c;
    }
}

What is the equals method?
suppose you have a structure containing objects of type C
you want to intern them: that is, have one object for each value
so you write this code, but it won’t work (why not?)

```java
public class C {
    private String s;
    public static Map<C,C> allocated = new ListMap<C,C>();
    public C intern () {
        C c = allocated.get(this);
        if (c == null) {
            allocated = allocated.put(this, this);
            return this;
        }
        return c;
    }
}
```

What is the equals method?
approaches

the problem: one equals method

- if it compares references with ==, then lookup won’t find match
- if it compares values, then interning is pointless!

have collection take equality predicate as argument

- can’t use standard Java collections: will have to make your own
- but see use of comparator objects in ordered types like java.util.TreeSet

use component as key instead of whole object

- eg, allocated maps String to C

for key, make wrapper around C object with its own equals

- not terrible, but a bit ugly
THE PERILS OF MUTATION
Mutation and equality

Two objects are equal now but different later

This can wreak havoc with data-structure invariants!
Ex: hash map structure

representation

- array of bucket lists

```java
class HashMap <K,V> {
    Entry<K,V>[] table;
    class Entry<K, V> { K key; V val; Entry<K,V> next; ... }
```
hash map operations

operations

- put\((k,v)\): to associate value \(v\) with key \(k\)

- get\((k)\): to get value associated with key \(k\)

resizing

- if map gets too big, create new array of twice the size and rehash
why does hashing work?

- rep invariant: entries are in buckets indexed by hash
  - all $i$ in $\text{table.indexes}$, $e$ in $\text{table}[i].\text{next}^*$ | $\text{hash}(e.\text{key}) == i$

- from object contract: equal keys have equal hashes
  - all $k, k'$: $\text{Key}$ | $k.\text{equals}(k') \Rightarrow \text{hash}(k) == \text{hash}(k')$

- consequence: need only look at one index
  - all $k$: $\text{Key}$, $i$: $\text{table.indexes}$ | $i != \text{hash}(k) \Rightarrow$ all $e$: $\text{table}[i].\text{next}^*$ | !$e.\text{key}.\text{equals}(k)$

Additional rep invariant: only one entry per key

- consequence: can stop at first match
mutating keys

what happens if you mutate a hash map’s key?

if equals and hashCode depend only on key’s identity
➢ nothing bad happens

if equals and hashCode depend on key’s fields
➢ then value of hashCode can change
➢ rep invariant of hash map is violated
➢ lookup may fail to find key, even if one exists

problem is example of ‘abstract aliasing’
➢ hash map and key are “aliased”
public class BrokenHash {
    static class Counter {
        int i;
        void incr () {i++;}
        @Override public boolean equals (Object o) {
            if (!(o instanceof Counter)) return false;
            Counter c = (Counter) o;
            return c.i == i;
        }
        @Override public int hashCode () {return i;}
    }

    public static void main (String[] args) {
        Set m = new HashSet <Counter> ();
        Counter c = new Counter();
        m.add(c);
        System.out.println("m contains c: " + (m.contains(c) ? "yes" : "no"));
        c.incr();
        System.out.println("m contains c: " + (m.contains(c) ? "yes" : "no"));
    }
}
so what to do?

**option #1 (Liskov)**
- equals on mutable types compares references
- no problem with keys, but two sets with same elements are not equal

**option #2 (Java Collections)**
- equals on mutable types compares current values
- forbid modification of objects held as keys
- more convenient for comparing collections, but dangerous