Mutability

Rob Miller
Fall 2009

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);
```

and how about this?

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

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

heap semantics of Java
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 `element_i` 'field' of a point to object `y`

null and primitives

**primitive values**
- eg, integers, booleans, chars
- are immutable (and aren’t objects)
- so whether shared is not observable

**null**
- is a value of object type
- but does not denote an object
- cannot call method on null, or get/set field

the operator `==`

**the operator `==`**
- returns true when its arguments denote the same object (or both evaluate to null)

**for mutable objects**
- if `x == y` is false, objects `x` and `y` are observably different
- mutation through `x` is not visible through `y`

**for immutable objects**
- if `x == y` is false, objects `x` and `y` might not be observably different
- in that case, can replace `x` by `y` and save space (called 'interning')
- Java does this with `Strings`, with unpredictable results
- lesson: don’t use `==` on immutables (unless you’re doing your own interning)

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

exercise: what’s wrong with this code? (hint: think about rep invariant)

```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];
                size--;
            }
    }
}
```

mutable datatypes

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);
    ...
}
```

classic mutable types

<table>
<thead>
<tr>
<th>interface in java.util</th>
<th>principal implementations</th>
<th>key mutators</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>ArrayList, LinkedList</td>
<td>add, set</td>
</tr>
<tr>
<td>Set</td>
<td>HashSet, TreeSet</td>
<td>add, remove, addAll, removeAll</td>
</tr>
<tr>
<td>Map</td>
<td>HashMap, TreeMap</td>
<td>put</td>
</tr>
</tbody>
</table>
how to pick a rep

lists
- use ArrayList unless you want insertions in the middle

sets and maps
- hashing implementations: constant time
- tree implementations: logarithmic time
- use hashing implementations unless you want determinism
- we’ll see later in this lecture how non-determinism arises

equality revisited

equivalence
- can define your own equality notion
  - but is any spec reasonable?

reasonable equality predicates
- define objects to be equal when they represent the same abstract value

a simple theorem
- if we define \( a \approx b \) when \( f(a) = f(b) \) for some function \( f \)
- then the predicate \( \approx \) will be an equivalence

an equivalence relation is one that is
- reflexive: \( a \approx a \)
- symmetric: \( a \approx b \Rightarrow b \approx a \)
- transitive: \( a \approx b \land b \approx c \Rightarrow a \approx c \)

the object contract

every class implicitly extends Object
- two fundamental methods:
  ```java
class Object {
  boolean equals (Object o) {...}
  int hashCode () {...}
  ...
}
```

“Object contract”: a spec for equals and hashCode
- equals is an equivalence (reflexive, symmetric, transitive)
- equals is consistent: if \( x.equals(y) \) now, \( x.equals(y) \) later
- hashCode respects equality:
  - \( x.equals(y) \) implies \( x.hashCode() = y.hashCode() \)
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

here's our first broken equality method

• violates transitivity: easy to see why

```java
public class Duration {
    private final int hours;
    private final int mins;

    public Duration(int h, int h) {
        hours = h;
        mins = m;
    }

    public boolean equals(Duration d) {
        // problematic, see next slide
        if (d == null) return false;
        return Math.abs(d.getMins() - this.getMins()) < CLOCK_SKEW;
    }
}
```

bug #2

what happens if you fail to override equals correctly

• note that outcome depends on declaration, not runtime type (aagh!)

```java
public class Duration {
    private final int hours;
    private final int mins;

    public Duration(int h, int h) {
        hours = h;
        mins = m;
    }

    public boolean equals(Duration d) {
        return d.getMins() == this.getMins();
    }
}
```

```java
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

public class Object {
  public boolean equals (Object o) {return o == this;}
}
public class Duration extends Object {
  public boolean equals (Object o) {return o == this;}
  public boolean equals (Duration d) {
    return d.getMins() == this.getMins();
  }
}

fixing equals

here’s a fix to the problem
• compile-time declaration no longer affects equality

@Override // compile error if doesn’t override superclass method
public boolean equals(Object o) {
  if (! (o instanceof Duration))
    return false;
  Duration d = (Duration) o;
  return d.getMins() == this.getMins();
}

equality and subclassing

now considering extending the type
• how should equality be determined?
• can’t rely on inherited equals method, because seconds ignored

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;}
  ...
}

bug #3

an attempt at writing equals for subclass

@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)); // false
System.out.println(d2.equals(d1)); // true
bug #4

yet another attempt

- this time not transitive

```java
@override public boolean equals(Object o) {
    if (! (o instanceof Duration)) return false;
    if (! (o instanceof ShortDuration)) return super.equals (o);
    ShortDuration d = (ShortDuration) o;
    return d.getSecs () == this.getSecs();
}
```

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)); // true
System.out.println(d2.equals(d3)); // false
System.out.println(d1.equals(d3)); // false!

solving the subclassing snag

no really satisfactory solution

superclass equality rejects subclass objects

- can write this
  ```java
  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

hash maps

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 structure

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td></td>
</tr>
</tbody>
</table>
```

© Robert Miller 2009

25

26

© Robert Miller 2009

26
hash map operations

operations
- put(k,v): to associate value v with key k
  - compute index i = hash(k)
  - hash(k) = k.hashCode & table.length-1 (eg)
  - if find entry in table[i] with key equal to k, replace val by v
  - otherwise add new entry for (k,v)
- get(k): to get value associated with key k
  - examine all entries in table[i] as for insertion
  - if find one with key equal to k, return val
  - else return null

resizing
- if map gets too big, create new array of twice the size and rehash

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

hashing principle

why does hashing work?
- rep invariant: entries are in buckets indexed by hash
  - all i: table.indexes, e: table[i].next | hash(e.key) == i
- from object contract: equal keys have equal hashes
  - all k, k': Key | k.equals(k') ⇒ hash(k) == hash(k')
- consequence: need only look at one index
  - all k: Key, i: table.indexes | i != hash(k) ⇒ all e: table[i].next | e.key.equals(k)
- also additional rep invariant: only one entry per key
- consequence: can stop at first match
- finally, keep buckets to small constant number of entries
  - then put and get will be constant time

e: table[i].next means e ranges over set of all entries reachable from table[i] in zero or more applications of next

example

what does this print?
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"));
    }
}

© Robert Miller 2009
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

**is Java consistent?**
- Object contract in Java says
  - It is *consistent* for any reference values \( x \) and \( y \), multiple invocations of \( x.equals(y) \) consistently return \( \text{true} \) or consistently return \( \text{false} \), provided no information used in \( equals \) comparisons on the object is modified

non-determinism

**to iterate over elements of a hash set**
- use `HashSet.iterator()`
- elements yielded in unspecified order

**what determines order?**
- code iterates over table indices
- so order related to hashing function
- depends on hash code, thus (for mutables) on object *addresses*

**so this means**
- different program runs likely to give different order
- this can be a real nuisance: consider regression testing, for example
- solution: use a `TreeSet` instead

summary

principles

**object heap is a graph**
- to understand mutation & aliasing, can’t think in terms of *values*

**equality is user-defined but constrained**
- must be consistent and an equivalence

**abstract aliasing complicates**
- may even break rep invariant (eg, mutating hash key)