Six.005
Elements of Software Construction

Basics of Mutable Types
(lecture 16)

Daniel Jackson
April 14, 2007
topics for today

in preparation

• for implementing the photo organizer on Weds
• we need to understand more about mutable types

today

• heap semantics
• mutable types
• equality revisited
• hash maps
heap semantics of Java
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");
sb.append(" world");
System.out.println (sb);
StringBuffer sb2 = sb;
sb2.append ("!");
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
how mutation happens

through field setting

\cdot statement \texttt{x.f = y} makes \texttt{f} field of \texttt{x} point to object \texttt{y}

through array update

\cdot statement \texttt{a[i] = y} makes \texttt{element_i} ‘field’ of \texttt{a} point to object \texttt{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 ==

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
  \> 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

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

StringBuffer is a mutable datatype

' computation gives new values to existing objects with mutators

```
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

concurrency

• none of these are thread-safe
• if using with concurrent clients, must synchronize clients yourself
• if you want concurrency in operations, use java.util.concurrent versions
equality revisited
the object contract

every class implicitly extends **Object**

- two fundamental methods:

```
class Object {
    boolean equals (Object o) {...}
    int hashCode () {...}
    ...
}
```

“Object contract”: a spec for **equals** and **hashCode**

- **equals** is an equivalence (reflexive, symmetric, transitive) -- see Lecture 12
- **equals** is consistent: if `x.equals(y)` now, `x.equals(y)` later
- **hashCode** respects equality:
  ```
  x.equals(y) implies x.hashCode() = y.hashCode()
  ```
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: $a \approx b \land b \approx c \Rightarrow a \approx c$
equality for mutable types

when are two objects equal?

for immutable datatypes
\· when they represent the same value
\· just compare contents

for mutable datatypes
\· equality must be **consistent**: once equal, always equal
\· so equals should compare object identities (with ==)

Java Collection Framework view
\· equality is consistent **only** if no modifications occur that affect equality
\· convenient but dangerous (more on this in next section)
hash maps
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
  
  compute index $i = \text{hash}(k)$
  
  $\text{hash}(k) = k.\text{hashCode} \& \text{table.length-1}$ (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
hashing principle

why does hashing work?

• rep invariant: entries are in buckets indexed by hash
  \[ \text{all } i: \text{table.indexes } | \text{hash(table}[i].\text{key}) == i \]

• from object contract: if distinct hash, must be unequal
  \[ \text{all } k, k': \text{Key } | \text{k.equals(k')} \text{ implies hash(k)} == \text{hash(k')} \]

• consequence: need only look at one index
  \[ \text{all } k: \text{Key}, i: \text{int } | \text{i != hash(k)} \text{ implies all } e: \text{table}[i].\text{next } | \text{!e.key.equals(k)} \]

• 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
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
what does this print?

```java
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"));
    }
}
```
**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

the Java heap
- is a labelled graph: field is set of edges
- field setting changes target of an edge

mutable datatypes
- classic types: sequence, set, map
- equality should be invariant

hash maps
- assume that keys obey the Object Contract
- trouble if keys are mutated and equality not invariant
- non-deterministic iterators can be a problem
lecture exercise

a small investigation

• the spec of `java.util.Set` says that a set should not contain itself
• write a code snippet that violates this and goes badly wrong
• explain what’s happening carefully and succinctly
• how would applying this lecture’s concepts help?