basics of mutable types

Fall 2010
Recall 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()
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: $a \approx b \land b \approx c \Rightarrow a \approx c$
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 h) {hours = h; mins = m;}
    public int getMins() {return hours*60 + mins;}
}
```
Concrete and Abstract Values

Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 3);
Duration d3 = new Duration (0, 62);
Here’s our first broken equality method

- What is wrong with it?

```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;
    }
}
```

Violates transitivity
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 does this print?

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

Prints true

What does this print?

Object d1 = new Duration(1,2);
Object d2 = new Duration(1,2);
System.out.println(d1.equals(d2));

Prints false
What’s going on?

- We’ve failed to override `Object.equals`
- Method is chosen using compile-time type not runtime type
- Method has been overloaded, not overridden

```java
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();
    }
}
```
here’s a fix to the problem

- compile-time declaration no longer affects equality

```java
@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();
}
```
Mutable and Immutable Objects
pop quiz

what happens when this code is executed?

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

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

and how about this?

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

```java
StringBuffer sb2 = sb;
sb2.append ("!");
System.out.println (sb);
```
```
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);
    ...}
```
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 $y$
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
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];
                size--;
            }
        }
    }
}
```

What if `elements[size-1]` points to something that is equal to `o`?

While `elements[x]` holds a reference to an `Object` `obj`, `obj` will not be garbage collected, even if `x > size`. 
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, sometimes with unpredictable results!
(Immutable) Strings and Equality?

Immutable containers, e.g., strings

- What happens if we use == for the equals method?

  ```java
  String s1 = new String("hello");
  String s2 = new String("HELLO");
  What is s2 == s1.toUpperCase() ?  FALSE
  
  String s1 = new String("hello");
  String s2 = new String("hello");
  What is s2 == s1 ?  FALSE
  
  String s1 = "hello";
  String s2 = "hello"
  What is s2 == s1 ?  TRUE, because Java interns strings
  ```
Strings `x`, `y`, `z` all contain the string “123” and are `.equals()`.

Java interns literal strings, like “123”, and compile-time constant strings, like “1”+ “2” + “3”. Thus, `x` and `y` are interned to the same location and `x == y`. However, `z` is not automatically interned by Java and so `x != z`, i.e., `x == z` returns false.

The String class has a `.intern()` method which returns an interned version of the string, e.g., `x == z.intern()`. But one should avoid using `==` on strings in Java.
When are two lists equal?

- One answer: *When they contain equal elements in the same order*

Dinner party example: Set of tables, list of friends at each table

```java
List t1 = new LinkedList();
List t2 = new LinkedList();
Set s = new ArraySet();
s.add(t1);
s.add(t2);
// Now add friends to t1 and t2
```
```
t1.add(f1); t1.add(f2); t2.add(f3); t2.add(f4);
```

- What happens here? *Set* uses the equals method on its elements to detect duplicates and will only add *t1*. List *t2* will not be added since *t1* and *t2* are both empty lists and therefore equal.
(Mutable) Lists and Equality?

When are two lists equal?

- Another answer: *When they contain the same elements in the same order*
- That is, Set uses `==` to determine equality of contained elements

Dinner party example will "work"

```java
List t1 = new LinkedList(); List t2 = new LinkedList();
Set s = new ArraySet();
s.add(t1); s.add(t2);
// Now add friends to t1 and t2
f1.add(f1); t1.add(f2); t2.add(f3); t2.add(f4);
```

- What happens if Set contains a list of strings?

  *Set may contain duplicate strings – you depend on Java’s interning capability to eliminate duplicates.*
In the Java List specification, *two lists are considered equal if they contain the same elements in the same order, or equal elements in the same order*.

equals is called recursively on the elements.

What does that mean for hashCode?

What happens here?

```
List l = new LinkedList();
l.add (l);
int h = l.hashCode();
```

Java Collections API: While it is permissible for lists to contain themselves as elements, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a list.
## 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**
  - How does non-determinism occur?
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