6.170 Recitation 4

6.170 - Recitation 4 Topics Covered

- Object Equality
- Hashing
- Equality Problems in the Java API

Object Equality

Each ADT should provide its own notion of what it means for two instances of that ADT to be equivalent. In general, you should never use the `==` operator to compare two object references, unless you are explictly trying to see if they are the exact same object in memory.

Question: When would you do that?

- When there is only one copy of the object. We only know this will be true in the case of Java primitives and classes that guarantee only one instance of the class exists (such as `enum` classes and Singletons, more in the design patterns lectures).
- May be appropriate for mutable objects.

The `equals()` method specified in `Object` is used to test for reference equality. Each ADT you specify should override the `equals()` method, if needed, to provide a notion of equivalence for that ADT.

```
public boolean equals(Object obj)
```

The `equals()` method has the following specified requirements for non-null references:

- Never equal to null: x.equals(null) is always false.
- Reflexive: x.equals(x) is always true.
- Symmetric: If x.equals(y) it must be that y.equals(x)
- Transitive: If x.equals(y) and y.equals(z) then it must be that x.equals(z)
- Consistent: Multiple invocations of x.equals(y) return the same value unless one of the objects is mutated.

Note that this is quite an intentionally incomplete definition, and so the following meets the spec of being an equivalence relationship in Java, even though it does not seem particularly useful:

```
public boolean equals (Object obj) {
      return (obj instanceof ThisClass);
}
```

Question: Why does `equals(null)` return false?

Given that the method signature is:

```
public boolean equals(Object o)
```

The invocation `equals(null)` can do one of the following things:

1. Return true
2. Throw an Exception
3. Return false

Answer:

```
Option 1: Return true

If equals(null) were to return true, then intuitively, this would not
make any sense because a non-null object would be considered equal to
something that is not even an object. This is inconsistent with our
idea of equality.

Option 2: Throw an Exception

Because equals() is a method of Object, it is a fundamental method in
Java that is used often, and in a variety of ways. Thus, it would be
inconvenient if it threw an exception when passed a null reference
when there is a more reasonable response, namely:

Option 3: Return false

This consistent with our idea of equality because it considers
objects and non-objects unequal, so this appears to be the most
reasonable result for equals(null).
```

Hashing

A closely related notion to equality in Java is that of the object's hashcode. Hashcodes are used to store objects in hash-table data structures.

The goal of hash tables is to let us store objects in such a way that we can perform (expected) constant time lookups later on. We start with an array of ***buckets***. Note that it takes constant time to access any arbitrary bucket. A ***hash function*** is used to mathematically map an object to a bucket. The hash function is a mapping from an object (such as a `String`) to a bucket number. When an object is being stored, it is mapped to the appropriate bucket and stored there. When we are looking up an object, we only need to look at the bucket identified by that object's ***hash***. It is normal for more than one object to hash to the same bucket. This can be solved, by having a linked list for each bucket in the hash table. If a ***good enough*** hash function is chosen these lists will be small for each bucket and the time to look up an object will be roughly constant.

Java avoids the theoretical issues associated with forming well-balanced hash tables. The `hashCode()` method defined in `Object` lets any hash table implementation map an object to a number. The hash table code can then use an internal mapping to create the actual hash (i.e. bucket index) from this number.

Java really just has one main requirement for `hashCode()` implementations: whenever two objects are `equal()` they must have the same `hashCode()`. In general, whenever you override `equals()`, you will also need to override `hashCode()`.

Question: What would happen if this requirement was not maintained?

Answer:

```
Two "equal" objects could hash to different buckets, allowing
duplicates to unintentially be stored and breaking methods such as
contains.
```

Question: What about mutable objects and hashing?

Answer:

```
If the hashCode() method of the object depends on a mutable field,
bad things can happen. If you mutate the object after storing it you
won't be able to find it anymore because it will be in the wrong
bucket.
```

Example

The Java implementations of the `List` interface can hold mutable objects without any complications. However, `Set`'s are a special case. According to the specifications, the behavior of `Set` implementations can be completely arbitrary if an element contained in the set is mutated.

Question: Can you guess what the following does?

```
Set<List<String>> set = new HashSet<List<String>>();
List<String> list = new ArrayList<String>();
list.add("6.170");
list.add("6.004");
list.add("6.003");
set.add(list);
System.out.println(set.contains(list));
list.add("6.033");
System.out.println(set.contains(list));
set.add(list);
System.out.println("set has " + set.size() + " elements");
for (List<String> l : set) {
     System.out.println(list);
}
```

Answer:

```
true
false
set has 2 elements
[6.170 6.004 6.003 6.033]
[6.170 6.004 6.003 6.033]
```

Why exactly does this fail? You need to know some details of the implementations of both `ArrayList` and `HashSet` to figure it out.

`ArrayList.hashCode()`: The returned hashcode is based on the `hashCode()`'s of **all** the elements in the list. The implementation in `AbstractList` actually iterates over the entire list adding up the hashcodes.

`HashSet.contains(elt)`: Uses `elthashCode()` to map to a bucket in its internal hashtable. If it sees an object equal to `obj`, this method returns `true`, otherwise it returns `false`. Doesn't look anywhere else in the data structure.

`HashSet.add(elt)`: Maps the object to a bucket and adds it to the bucket if it isn't already in that bucket.

It is the changed `hashCode()` value that is the problem. If we had added a `StringBuilder` to the set instead, mutated it, and added it again, the set would have worked fine, since `StringBuilder` uses `Object.hashCode()`.

Equality Problems in the Java API

The idea of equality is confusing when dealing with mutable ADTs. The question to ask is, if two objects are equal now, will they always be equal in the future? For Java objects, the answer is "you choose" - the Object contract does not specify. For immutable objects, it is natural for equality to be eternal, but for mutable objects, there is a choice to be made here.

Question: What does the following print out?

```
StringBuffer s1 = new StringBuffer("hello");
StringBuffer s2 = new StringBuffer("hello");
System.out.println(s1.equals(s2));
```

Answer:

```
false (StringBuffer uses Object.equals())
```

Of course, the designers could have chosen differently, and made equals return true in such situations. But then equality would not be eternal. Two `StringBuffer` objects with the same content now may not have the same content in future. We see this other transient kind of equality implemented with Java's `Date` class, where equality is based on a elt-by-elt comparison of the objects' contents. However, here we run into another problem.

```
Date d1 = new Date(0); // January 1, 1970 00:00:00 GMT
Date d2 = new Date(0);
System.out.println(d1.equals(d2));
d2.setTime(1); // a millisecond later
System.out.println(d1.equals(d2));
```

prints:

```
true
false
```

Unfortunately, Java does not follow a consistent notion of equivalence for mutable ADT's, beyond the specifications in `Object`. In general, you should look at the specs of the class you are using before relying on the equals method of the class.

Question: What does the following print:

```
List<String> arrayList = new ArrayList<String>();
List<String> linkedList = new LinkedList<String>();
arrayList.add("a");
linkedList.add("A".toLowerCase());
arrayList.add("b");
linkedList.add("B".toLowerCase());

System.out.println(arrayList.equals(linkedList));
System.out.println(linkedList.equals(arrayList));
```

Answer:

```
true
true
```

The classes in the Java collections framework consistently use the notion of equivalence as defined by the elements in the collection. This is why the `LinkedList` above is ***equal*** to the `ArrayList`. Furthermore, note that the references in the two lists are not `==` to each other.

In the Java `List` specification, two lists are equal not only if they contain the same elements in the same order, but also if they contain equals elements in the same order. In other words, the equals method is called recursively. To maintain the Object contract, the `hashCode` method is also called recursively on the elements. The algorithm used to calculate hashcode for `List` is actually specified in detail as:

```
hashCode = 1;
Iterator i = list.iterator();
while(i.hasNext()) {
  Object obj = i.next();
  hashCode = 31*hashCode + (obj==null ? 0 : obj.hashCode);
}
```

This is verbatim from the Java specification. Why is code showing up in a specification? Does the client care about how hashCode is implemented?

Possible answers:

```
Con: Giving the code commits Java to that code.
Con: Better hashing techniques cannot be added without causing pain.
Con: If the implementation is bad, the consequences are amplified.

Pro:  Clients knows whether their use of hashes will be efficient.
```

Java classes generally give the details of their hashCode implementation, and are worth looking at as examples of how such methods are written in practice.

The recursion in `hashCode` results in a very nasty surprise. The following code, in which a list is inserted into itself, will actually fail to terminate!

```
List<List> lst = new LinkedList<List>();
lst.add(lst);  // okay so far...
int h = lst.hashCode();  // infinite loop!
```

This is why you'll find warnings in the Java API documentation about inserting contains into themselves, such as this comment in the specification of List:

```
Note: 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.
```

Lesson:

Intuitions about equality can be inconsistent: if you don't think through your approach to object identity it can be very hard to fix problems that arise later. The simplest and most reliable way to treate equality is to use reference equality on mutable types, and that you should always consider that first. Sometimes, in the case of Collections, you may want to use the other approach, but it's best avoided. It's generally possible to use override the `equals()` method and use the Java collection classes without problems, so long as you're careful.

Java Puzzler: EXCEPTIONALLY ARCANE

What do these programs do?

1)

```
import java.io.IOException;
```

```
public class Arcane1 {
  public static void main(String[] args) {
    try {
      System.out.println("Hello world");
    } catch (IOException e)
      System.out.println("I've never seen println fail!");
    }
  }
}
```

Does not compile. The language specification says that it is a compile-time error for a catch clause to catch a checked exception type E if the corresponding try clause can't throw an exception of some subtype of E.

2)

```
public class Arcane2 {
  public static void main(String[] args) {
    try {
      // If you have nothing nice to say, say nothing
    } catch (Exception e) {
      System.out.println ("This can't happen.");
    }
  }
}
```

Prints nothing. Catch clauses that catch Exception or Throwable are legal regardless of the contents of the corresponding try clause.

3)

```
interface Type1 {
  void f() throws CloneNotSupportedException;
}
interface Type2 {
  void f() throws InterruptedException;
}
interface Type3 extends Type1, Type2 {
}
public class Arcane3 implements Type3 {
  public void f() {
    System.out.println("Hello world");
  }
  public static void main(String[] args) {
    Type3 t3 = new Arcane3();
    t3.f ();
  }
}
```

Prints "Hello world". It is not a compile error for the interfaces to throw different exceptions, and it is not a compile error to catch neither. The set of checked exceptions that a method can throw is the intersection of the sets of checked exceptions that it is declared to throw in all applicable types, not the union. As a result, the f method on an object whose static type is Type3 can't thorw any checked exceptions at all.