CS310 - Advanced Data Structures and Algorithms

Spring 2014 – Class 4

February 6, 2014

Announcements

- Remaining quals returned today
- Solutions available online
- See UML diagram handout for class structure

Notes on Removing Duplicates From a List

```
 List<Order> list = new LinkedList<Order>(); // or ArrayList
// add some elements to the list:  Foo, Bar, Bar, Bam
Iterator<Order> itr = list.iterator();
int position = 0;
System.out.println("about to do next() in outer loop...  list is"
+list);
Order o1 = itr.next(); // throws ConcurrentModificationException
here, after first remove
System.out.println(``working on '' + o1);
ListIterator<Order> listItr = list.listIterator(position + 1);
while (listItr.hasNext()) {
    Order o2 = listItr.next();
    System.out.println("inner loop o2 =" + o2);
    if (o1.equals(o2)) {
        System.out.println("removing o2 =" + o2);
        listItr.remove();
    }
}
position++;
```

Notes on Removing Duplicates From a List

Output:

List with Foo, Bar, Bar, Bam:
about to do next() in outer loop... list is[Foo, Bar, Bar, Bam]
working on Foo
inner loop o2 =Bar
inner loop o2 =Bar
inner loop o2 =Bam
about to do next() in outer loop... list is[Foo, Bar, Bar, Bam]
working on Bar
inner loop o2 =Bar
removing o2 =Bar
inner loop o2 =Bam
about to do next() in outer loop... list is[Foo, Bar, Bam]
Exception in thread "main" java.util.ConcurrentModificationException
at java.util.AbstractList$Itr.checkForComodification(Unknown Source)
at java.util.AbstractList$Itr.next(Unknown Source)
at ListTest.main(ListTest.java:20)

Order Class

- We need to override equals to have proper duplicates in the list, which can be found using .equals.
- If we remove the override of equals in Order, there will be no duplicates in the list if we separately create each element Order, because each element will have a different reference value.

```
public class Order {
    private String foo;
    public Order(String bar) {foo = bar;}
    public String toString() { return foo; }
    @Override
    public boolean equals(Object x) {
        if (x==null || getClass() != x.getClass())
            return false;
        Order o = (Order)x;
        return foo.equals(o.foo); // using String equals
    }
}
```

Safely Removing Duplicates

- Drop one iterator, leaving position, get o1 by get(position), though this does lower potential performance with LinkedList, and remove is a similar problem in ArrayList
- What to do with huge lists, when using get and/or remove in inner loop means $O(n^2)$ or worse?
- Abandon lists!
- You can use HashSet h = new HashSet(list); //Set means no dups, $O(n)$
- Then put result back in a list.
- Another way: toArray, then sort, then pick off unique values, $O(n \log n)$

Introduction to Sets

- A set contains a number of elements, with no duplicates and no order.
- A = {1, 5, 3, 96}, or B = {17, 5, 1, 96}, C={"Mary", "contrary", "quite"}.
- Incorrect - {"Mary", "contrary", "quite", "Mary"}.
- In Java, the Set interface is the Collection interface.
- The API isnt sensitive to the lack of duplicates, only the implementation. The implementations in the JDK are the TreeSet and HashSet.
- They check for duplicates by using the equals method of the elements.
- HashSet uses hashCode() and TreeSet uses compareTo().

TreeSet Example

```
public static void main(String[] args)
{
  Set<String> s = new TreeSet<String>();
  // For reverse order:
  // TreeSet<String>(Collections.reverseOrder());
  s.add(``joe'');
  s.add(``bob'');
  s.add(``hal'');
  printCollection(s);
}
```

Sets of JDK Element Type

- Sets of type String, Integer, etc. are very easy to use, JDK classes all have appropriate equals, hashCode, and compareTo (they implement Comparable<E>).
- Ex: Simple set app using element type String.
- If we add "hal" again, no difference in resulting Set.

```
public static void
main(String[] args)
{
  Set<String> s =
    new HashSet<String>();
  s.add(``joe'');
  s.add(``bob'');
  s.add(``hal'');
  printCollection(s);
}
```

Sets of JDK Element Type

- Note that a pure set is supposed to be without order, and here we are seeing order imposed by the TreeSet.
- It's an extra feature, so the TreeSet gives us a SortedSet.
- We can just ignore the order if we want.
- The TreeSet gives a high-performance implementation, competitive with HashSet.

Sets of User Defined Objects

- If we use our own class for the element type, we have to make sure equals, and hashCode or compareTo are in good enough shape to work properly when called by HashSet or TreeSet on the element objects.
- Consistency requirements: equals and hashCode must be consistent, so that if a.equals(b), then a.hashCode() == b.hashCode().
- Also equals and compareTo must be consistent, so that if a.equals(b), then a.compareTo(b) == 0.

Equals Example – Potentially Problematic

```
class BaseClass {
 public BaseClass(int i)
   {x = i;}
 public boolean equals(Object rhs)
 {
   if(rhs == null ||
       getClass() !=
       rhs.getClass())
     return false;
   return x ==
       ((BaseClass) rhs)).x;
 }
 private int x;
}
```

```
 class DerivedClass extends
BaseClass {
  public DerivedClass(int i, int j)
    { super( i ); y = j; }
  public boolean equals(Object rhs)
  {
    return super.equals(rhs) &&
        y == ((DerivedClass) rhs).y;
  }
  private int y;
}
```

What if we substituted the comparison by : if (! rhs.instaceof(BaseClass) return false; ?

Equals Example

- Question: is BaseClass in good shape to make HashSet<BaseClass>?
- Answer: No, it overrides equals, but not hashCode.
- If it overrode neither, it would be OK, but once equals is overridden, you need to override hashCode too, and consistently.
- Fix: drop equals(), or add:

```
public int hashCode()
{ return Integer(x).hashCode(); }
```

Equals Example

- Question: is BaseClass in good shape to make "new TreeSet<BaseClass>()"?
- Answer: No, it doesn't implement Comparable<BaseClass>.
- Fix: add "implements Comparable<BaseClass>", and method compareTo:

```
public int compareTo(BaseClass b)
{ return Integer(x).compareTo(b.x); }
```

Student Example

```
 /* Students are ordered on basis of name only.  */
class SimpleStudent implements Comparable<SimpleStudent> {
  public SimpleStudent(String n, int i)
    { name = n; id = i; }
  public boolean equals(Object rhs) {
    if (rhs == null || getClass() != rhs.getClass())
      return false;
    SimpleStudent other = (SimpleStudent) rhs;
    return name.equals(other.name);
  }
  public int compareTo(SimpleStudent other)
    { return name.compareTo(other.name); }
  public int hashCode()
    { return name.hashCode(); }
}
```

equals/hashCode in User Defined Sets

- We can drop equals and hashCode from the class implementation.
- It will allow students with the same name.
- With the Object equals, two students with the same name ("Bob, 1") and ("Bob, 2") are different elements, so the set will contain 2 elements.
- If we use the equals that is coded above, the two "Bob" objects are .equals each other, and the set will contain only 1 element.

Equals and Compare's

- Need to fix up SimpleStudent, pg. 269:
- Doesn't implement the required compareTo, so add:
  ```
  int compareTo(SimpleStudent s)
  { return name.compareTo(s.name); }
  ```
- Unlike equals, use the non-Object type here.
- Consistent with equals: compareTo == 0 for equals objects.
- Make fields private, add getters (and setters if deemed appropriate).
- Now we have SimpleStudent in shape for both HashSet<SimpleStudent> and TreeSet<SimpleStudent>.

Another Set Example

- counting vowels. We want something like:

```
if (c is a vowel)
  count it
```

- How do we code the condition here?
- Suggested solution:

```
if (c == 'a'||c == 'e'||c == 'i'||c == 'o'||c ==
'u')
```

- or maybe:

```
String vowels = ``aeiou'';
if (vowels.indexOf(c) >= 0)
 // search string for c
```

Another Set Example

- We can also build a Set of vowels:

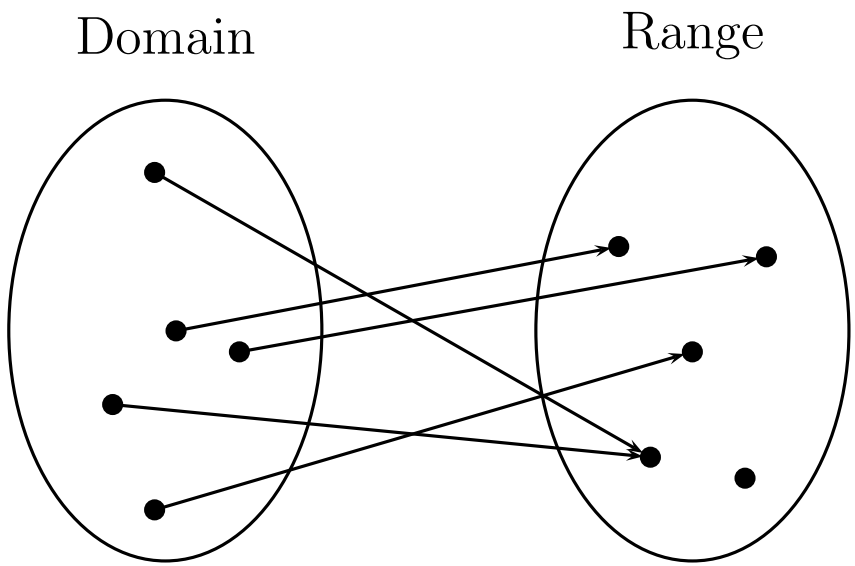
```
Set<Character> vowels = new HashSet<Character>();
// Autoboxing here:  char -> Character
vowels.add('a');
vowels.add('e');
vowels.add('i');
vowels.add('o');
vowels.add('u');
```

- In main loop over some string s: `c = s.charAt(i)`

```
if (vowels.contains(c))
 // assuming c a Character:  O(1) lookup
 count++;
```

Maps – Definition

- Given two sets, Domain and Range, with a relation from one to another.
- Like a math function, each domain element has associated with it exactly one range element.
- Two arrows can land on the same range element, but one domain element cannot have two arrows out of it.



Maps – Definition

- The action of following the arrow is often known as a "lookup" action.
- For ex., employee records are looked up by social-security no. and/or by employee name.
- Social security numbers or employee names are the Domain, Employee objects are the Range.
- In programming, Maps are lookup tables.
- We are mapping integers to employee objects or Strings to employee objects.
- Mapping creates a pair of <DomainType,RangeType>.
- The DomainType is a key, the RangeType is a value.

Maps – Cont.

- A simple example: Descriptions of grades:
  'A' $\rightarrow$ "excellent"
  'B' $\rightarrow$ "good"
  'C' $\rightarrow$ "ok"
- Here the DomainType is char and the RangeType is string.
- Each of these lines can be called a "key/value pair", or just "pair".
- ('A', "excellent") is a pair of the grade 'A' (the key) and the phrase "excellent" (the value).
- The whole mapping is the set of these 3 pairs.
- M = { ('A', "excellent"), ('B', "good"), ('C', "ok") } – a map as a set of pairs, or "associations"

Maps – Cont.

- A mapping is a collection like other collections we are studying, lists, stack, queues, and sets.
- However, in Java a Map has its own interface separate from Collection.
- Note that not every collection of pairs makes a proper map: M qualifies as a map only if the collection of keys has no duplicates, i.e., constitutes a set.

The Map Interface

```
// Map interface.
public interface Map<KeyType,ValueType> extends Serializable
{
  // Returns the number of keys in this map.
  int size( );
  // Tests if this map is empty.
  boolean isEmpty( );
  // Tests if this map contains a given key.
  boolean containsKey( KeyType key );
  // Returns the value in the map associated with the key.
  ValueType get( KeyType key );
  // Adds the key value pair to the map, overriding the
  // original value if the key was already present.
  ValueType put( KeyType key, ValueType value );
  // Remove the key and its value from the map.
  ValueType remove( KeyType key );
  // Removes all key value pairs from the map.
  void clear( );
  // Returns the keys in the map.
  Set<KeyType> keySet( );
  // Returns the values in the map.  There may be duplicates.
  Collection<ValueType> values( );
  // Return a set of Map.Entry objects corresponding to
  Set<Entry<KeyType,ValueType> > entrySet( );

}
```

The Map Interface

```
/**
* The interface used to access the key/value pairs in a map.
* From a map, use entrySet().iterator to obtain a iterator
* over a Set of pairs.  The next() method of this iterator
* yields objects of type Map.Entry.
*/
public interface Entry<KeyType,ValueType> extends Serializable
{
  // Obtains this pair's key.
  KeyType getKey( );
  // Obtains this pair's value.
  ValueType getValue( );
  // Change this pair's value.
  ValueType setValue( ValueType newValue );
}
```

Actions on Maps

- We can add a key/value pair to a Map, and this operation is called put in Java, and we can lookup the associated range element (value) of any given domain element (key), and this action is called get.
- Put is more pushy than "set" for Lists – it can put another fact into the collection rather than just changing one thats there.
- Like sets, Java supports two main implementations: TreeMap and HashMap.

Ways of Thinking About Maps

- As holding conversions, like codes to grades, social security number to name.
- As generalized arrays.
- As math functions: $y = f(x)$ is a map.
- As a "database" with key lookup: SSN to employee record, ISBN to book record, name to inventory record.

Map Example – Phone Numbers

```
 public static void demo( Map<String,String> phone1 )
{
  phone1.put( "John Doe", "212-555-1212" );
  phone1.put( "Jane Doe", "312-555-1212" );
  phone1.put( "Holly Doe", "213-555-1212" );
  phone1.put( "Susan Doe", "617-555-1212" );
  phone1.put( "Jane Doe", "unlisted" );
  System.out.println("phone1.get("Jane Doe"):  " + phone1.get("Jane
Doe"));
  Set<String> keys = phone1.keySet( );
  printCollection( keys );
  Collection<String> values = phone1.values( );
  printCollection( values );
  keys.remove( "John Doe" );
  values.remove( "unlisted" );
  System.out.println("After John Doe and unlisted are removed, the map
is");
  printMap( "phone1", phone1 );
  System.out.println( phone1 );
}
```

Map Application Example – Phone Numbers

- Can do simple lookups: phone1.get("John Doe");
- What happens if two people have the same name while doing puts?
- Answer: The second put overwrites the first.
- How to detect it?
- Answer: Look at the return value of put, the old value or null if nothing there.
- We could also map from phone number to name – just phone2.put(...)