132 H Midterm Answers, Spring 2011

Q1

Critique the following class. In particular, point out any aspects of the design or implementation of this class that violate properties that all Java classes should respect.

```
public class Point2D {

   public final int x,y;

   public Point2D(int x, int y) {
      this.x = x;
      this.y = y;
   }

   public boolean equals(Object o) {
      Point2D that = (Point2D) o;
      return this.x == that.x && this.y == that.y;
   }

   public String toString() {
      return String.format("(%d, %d)", x, y);
   }

}
```

Answer: The method violates the hashCode/equals contract: objects that compare as equal may have different hashCodes.

Also, the equals method throws an exception if it is passed null or something that isn't a Point2D. It should just return false in those cases.

Q2:
The following is part of a LinkedList implementation. Note that in this design, when the list is empty, the head field is null. Provide an implementation of the remove method.

```
public class LinkedList<E> {

    static class Node<E> {
        E value;
        Node<E> next;
    }

    Node<E> head = null;

    public boolean remove(Object value) {
       if (head == null)
          return false;
       if (head.value.equals(value)) {
          head = head.next;
          return true;
          }
        Node<E> n = head;
        while (n.next != null) {
           if (n.next.value.equals(value)) {
             n.next = n.next.next;
             return true;
             }
           n = n.next;
    }
   return false;
}
```

Q3.
Considering a typical linked list and array list implementation,

- Name an operation that would be more expensive with an array list than with a linked list.
  Answer: inserting an element at the beginning of the list

- Name an operation that would be more expensive with a linked list than an array list

  Answer: getting or setting an element at an arbitrary position in the list.

Q4.
Critique the following hashCode implementation for an implementation of Set<E>:

```
@Override
public int hashCode() {
    int result = 0;
    for (E e : this) {
        result = result * 37 + e.hashCode();
    }
    return result;
}
```

Answer: This makes the hashCode dependent on the order in which elements are iterated. Equality for Sets is determined only by the elements in the set, not the order in which they are iterated over. In particular, for something like a HashSet, this code could cause two sets with the same elements to have

different hashCodes(). Even resizing/rehashing a HashSet would likely cause the hashCode to change.

Q5.
Discuss possible data structures for implementing stacks and queues. How do stacks and queues differ in terms of the constraints/requirements for their implementation? Give an example of an implementation that would be appropriate for one but not the other.

Answer: Both involve a linear ordering of elements. A stack requires access to and modification of only one end of the order; a queue requires access to both ends.

A straightforward and efficient implementation of a stack is an array and an index of for the current top of stack, with the stack contents being stored in a[0..top]. When the stack grows larger than the size of the array, a new array, twice as large, is created and the old values are copied to it.

For a queue, the simple data structure is a doubly linked list, with pointers to the first and last element, allowing for quick addition or removal at either end (e.g., this could implement a dequeue as well as a queue). But since it using a separate object for each value in the queue, it isn't very space efficient. A more efficient data structure is a circulate queue, implemented using an array and indexes for the first and last element of the queue. The tricky thing here is that the array has to be treated as a circular buffer. For example, with an array of length 8 and 5 elements in the queue, the values of the queue might be stored in a[6], a[7], a[0], a[1] and a[2], with a[6] being the next value to be dequeued and a[2] the last value added to the queue.

Q7.
Below is part of a class that implements a range of integers. For example, the code:

```
for(int i : new Range(1,5))
        System.out.println(i);
```

should print five lines, 1, 2, 3, 4 and 5. Provide the iterator() method of Range.

```
public class Range implements Iterable<Integer> {

    public Range(int min, int max) {
        this.min = min;
        this.max = max;
    }

    final int min,max;

@Override
    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            int next = min;
            @Override
            public boolean hasNext() {
                return next <= max;
            }

            @Override
            public Integer next() {
                if (!hasNext())
                    throw new NoSuchElementException();
                return next++;
            }

            @Override
            public void remove() {
               throw new UnsupportedOperationException();
            }
        };
    }
```

Q8. Assume you have the following code:

```
public class Foo<E> {

    public void f1(Collection<? extends E> c) { … }
    public void f2(Collection<E> c) { … }
    public void f3(Collection<? super E> c) { … }


    public static void main(String args[]) {
        Foo<Number> foo = new Foo<Number>();
        Collection<Object> cObject = new ArrayList<Object>();
        Collection<Number> cNumber = new ArrayList<Number>();
        Collection<Integer> cInteger = new ArrayList<Integer>();

        // which values can be passed to which methods?
    }
}
```

Note that Integer is a subtype of Number.

For each of the following combinations of functions and arguments, write yes or no as to whether that value can be passed to that method.

	cObject	cNumber	cInteger
f1	no	yes	yes
f2	no	yes	no
f3	yes	yes	no

What can you tell what the functions f1, f2 and f3 do to their arguments by just looking at the method declarations?

- f1 – that it examines or uses values from the collection, but doesn't add to the collection
- f2 - nothing
- f3 – that it might add elements of type E to the collection, but if it examines elements in the collection, it can only treat them as Objects.