1. (14 points) Pages 7 to 9 define a simplified version of the Java Collection interface (the SimpleCollection and SimpleIterator interfaces), together with a concrete implementation using linked lists (the class LinkedSimpleCollection and the auxiliary class Node). These definitions are identical to those in the review handout, except that the LinkedSimpleIterator class has been slightly simplified.

Suppose we execute the following program:

```java
public class Main {
    public static void main(String[] args) {
        SimpleCollection<String> c = new LinkedSimpleCollection<String>();
        c.add("c"); c.add("b"); c.add("a");
        SimpleIterator<String> i = c.iterator();
        i.next();
        c.removeAll("b");
        String s = i.next(); // HERE
    }
}
```

In the space below, draw a diagram of the memory configuration (both stack and heap) when execution reaches the line marked HERE. (You can omit the parameter args to the main method.) Use the same conventions as the sample stack/heap drawing on page 10.

*Answer:*

```
Stack
  c
  i
  s

Heap
  LinkedSimpleCollection
      first
  Node
      element
      next
  Node
      element
      next
  String
      "a"
  Node
      element
      next
  String
      "b"
  Node
      element
      next
  String
      "c"
  LinkedSimpleIterator
      current
```

Grading scheme: 11-14 points for nearly correct answers; 7-10 points for answers with 2 or 3 individual mistakes; 3-6 points for answers with several issues.
2. (6 points) Briefly (in two or three sentences) compare the way mutable state is treated in OCaml and in Java.

Answer: OCaml uses several different language constructs to deal with different aspects of mutable state: naming (variable binding), mutability (ref types), and the possibility for a value to be “missing” (option types). Java combines these features: every variable is mutable, and every variable of reference type can contain the special value null. Grading scheme: 3 points for observing that Java defaults to mutable and OCaml defaults to immutable; 2 points for mentioning explicit ref types (or explicit ! and :=); 1 point for mentioning null vs. option.
3. (4 points) What gets printed if the expression \(\text{new} \ C().\text{foo()}\) is evaluated in the presence of the following class definitions?

```java
class MyExn extends RuntimeException {}

class AnotherExn extends MyExn {}

class C {
    public void foo() {
        try {
            System.out.println("a");
            bar();
            System.out.println("b");
        } catch (AnotherExn e) { System.out.println("c"); }
        catch (MyExn e) { System.out.println("d"); }
        finally { System.out.println("e"); }
    }
    public void bar() {
        try { baz(); } 
        catch (AnotherExn e) { System.out.println("f"); throw e; }
        catch (MyExn e) { System.out.println("g"); throw new AnotherExn(); }
        finally { System.out.println("h"); System.out.println("i"); }
    }
    public void baz() {
        try { System.out.println("j");
            if (true) throw new MyExn();
            System.out.println("k"); } 
        catch (AnotherExn e) { System.out.println("l"); }
        finally { System.out.println("m"); }
        System.out.println("n");
    }
}
```

Answer:

```
a
j
m
g
h
c
e
```
4. (14 points) Complete the following definition of the class IterMax. Instances of IterMax are iterator transformers—iterators built from other iterators. The IterMax constructor takes a String iterator wrapped (an instance of the SimpleIterator class defined on page 9) as an argument, and the hasNext and next methods use the corresponding methods of wrapped to do their job. Each time next is called on the outer IterMax iterator, it should return the first element from the inner wrapped iterator that is longer than all the elements previously returned. For example if the inner Iterator produces the sequence

"a" "bb" "c" "ddd" "ab" "abc" "eeee"

then the outer one would produce:

"a" "bb" "ddd" "eeee"

The outer iterator should never return null (even if the inner iterator does).

Answer:

class IterMax implements SimpleIterator<String> {  
    SimpleIterator<String> it;  
    int m = -1;  
    String n;  

    IterMax (SimpleIterator<String> wrapped) {  
        it = wrapped; advance();  
    }  

    public boolean hasNext () { return (n != null); }  

    public String next () {  
        String s = n;  
        advance();  
        return s;  
    }  

    void advance () {  
        boolean done = false;  
        n = null;  
        while (!done) {  
            if (!it.hasNext()) { done = true; }  
            else {  
                n = it.next();  
                if (n != null && n.length() > m) {  
                    done = true;  
                    m = n.length();  
                }  
            }  
        }  
    }  
}

5. (12 points) This problem again refers to the definitions of SimpleCollection and related classes on pages 7 to 9. Your job is to complete the definition of a method interleave that combines the contents of two LinkedSimpleCollections “in-place,” in alternating order. For example, if c contains "a", "c", and "e" (in that order) and c2 contains "b", and "d" (in that order), then after c.interleave(c2), c should contain "a", "b", "c", "d", and "e", and c2 should be empty. If the two initial collections are not the same size, the extra elements should appear at the end of the interleaved collection. Your solution should not create any new objects (i.e., don’t use new anywhere).

Answer:

```java
class LinkedSimpleCollection<E> implements SimpleCollection<E> {
    // ...
    public void interleave(LinkedSimpleCollection<E> other) {
        if (other == null) {
            return;
        }

        if (this.first == null) {
            this.first = other.first;
            other.first = null;
            return;
        }

        if (other.first == null) return;

        Node<E> last = this.first;
        Node<E> left = this.first.next;
        Node<E> right = other.first;
        other.first = null;
        boolean takeRight = true;

        while (true) {
            if (left == null) {
                last.next = right;
                return;
            } else if (right == null) {
                last.next = left;
                return;
            }

            if (takeRight) {
                last.next = right;
                right = right.next;
            } else {
                last.next = left;
                left = left.next;
            }

            if (takeRight) {
                takeRight = false;
                right = other.first;
                other.first = null;
            } else {
                takeRight = true;
            }
        }
    }
}
```
Grading scheme: The top level grading was broken into three 4-point sections: the basic pointer setup, the basic loop structure, and the corner cases necessary for correctness.

Single point deductions were made for, e.g., not considering `other == null`. Larger deductions were made for bad logic in the loop.

Solutions with iterators and some logic were typically awarded 2 points.
For Reference:
SimpleCollection, SimpleIterator, and Node

interface SimpleCollection<E> {
    boolean removeAll(E element);
    boolean add(E element);
    boolean contains(Object o);
    SimpleIterator<E> iterator();
}

interface SimpleIterator<E> {
    boolean hasNext();
    E next();
}

class Node<E> {
    public E element;
    public Node<E> next;

    public Node (E element, Node<E> next) {
        this.element = element;
        this.next = next;
    }
}
class LinkedSimpleCollection<E> implements SimpleCollection<E> {
    private Node<E> first = null;

    public boolean removeAll(E element) {
        boolean changed = false;
        while (first != null && first.element.equals(element)) {
            first = first.next;
            changed = true;
        }
        if (first == null) return changed;
        Node<E> prev = first;
        for (Node<E> current = first.next;
             current != null;
             current = current.next) {
            if (current.element.equals(element)) {
                prev.next = current.next;
                changed = true;
            }
            prev = prev.next;
        }
        return changed;
    }

    public boolean add(E element) {
        Node<E> newnode = new Node<E>(element, first);
        first = newnode;
        return true;
    }

    public boolean contains(Object o) {
        for (Node<E> current = first; current != null; current = current.next) {
            if ((current.element == null && o == null)
                || (current.element != null && current.element.equals(o))) {
                return true;
            }
        }
        return false;
    }

    public SimpleIterator<E> iterator() {
        return new LinkedSimpleIterator<E>(first);
    }
}
For Reference: LinkedSimpleIterator class

class LinkedSimpleIterator<E> implements SimpleIterator<E> {
    private Node<E> current;

    public LinkedSimpleIterator (Node<E> first) {
        current = first;
    }

    public boolean hasNext() {
        return (current != null);
    }

    public E next() {
        if (current == null) {
            throw new java.util.NoSuchElementException();
        } else {
            E n = current.element;
            current = current.next;
            return n;
        }
    }
}
For Reference: Sample Stack/Heap Drawing