Introduction to Computer Science II
Fall 2016
Final exam — Solutions

1. Questions: Provide short answers to each of the following questions:

   (a) Why are *compile-time errors* preferable to *run-time errors*?

   (b) Can an *instance* method call a *static* method? Can a *static* method call an *instance* method? If so, explain how; if not, explain why not.

   (c) What are the similarities and differences between an *interface* and an *abstract class*.

   Answers:

   (a) Compile-time errors are found before the program runs; when a program compiles without errors, a number of errors (e.g., type mismatches) are proven impossible. Run-time errors may occurs at any time during execution, and may be dependent on the inputs provided, making such errors more challenging to discover.

   (b) Any instance method can call a static method, but static methods may not call instance methods directly. However, a static method may call an instance method via an object that contains that instance method.

   (c) Both interfaces and abstract classes are not directly instantiable, because both specify abstract methods. However, interfaces are composed entirely of abstract methods and publicly accessible members, while abstract classes may have a few as one abstract method, as well as many private members. A class may implements multiple interfaces, but a class can extend only one abstract class.

DISCUSSION:

(a) Answers to this question were largely good. There was, amusingly, a great deal of tautological declaration (e.g., *"Compile-time errors occur when the program is compiled."*) More substantively, many people overlooked that a successful compilation implies a proof that the program cannot contain type mismatches.

(b) Most people explained well the ambiguity inherent in a static method's attempt to call an instance method. However, many people then overlooked that a static method could call an instance method *if it did so on a specific object*.

(c) The answers here were quite good. The most overlooked element was that a class can implement many interfaces, yet it can extend only one abstract class.

2. QUESTION: Consider the following interface:

```
public interface GrabBag<T> {

  // Drop a value into the bag.
  public void dropInto (T v);

  // Pull a randomly selected value from the bag.
  // If the bag is empty, return null.
  public T pullFrom ();

  // Return the number of values in the bag.
  public int size ();

}
```

(a) Write a class that implements this interface.

(b) Recall your `Deck` class from Project-2—specifically, recall the `shuffle()` method. Rewrite that method such that it uses a `GrabBag<Card>` to perform the shuffle.

ANSWER:

(a) An implementation of `GrabBag`...

```
import java.util.ArrayList;
import java.util.Random;

public class RealGrabBag <T> implements GrabBag <T> {

    private ArrayList<T> _bag;
    private Random _r;

    public RealGrabBag () {
        _bag = new ArrayList<T>();
        _r = new Random();
    }

    public void dropInto (T v) {
        _bag.add(v);
    }

    public T pullFrom () {
        return _bag.remove(_r.nextInt(_bag.size()));
    }

    public int size () {
        return _bag.size();
    }

}
```

(b) An implementation of the `shuffle()` method...

```
public class Deck {

   private Card[] _deck;
   private int _top;

   ...

   public void shuffle () {

      GrabBag<Card> bag = new RealGrabBag<Card>();
      for (int i = 0; i <= _top; i += 1) {
        bag.dropInto(_deck[i]);
      }

      for (int i = 0; i <= _top; i += 1) {
        _deck[i] = bag.pullFrom();
      }

   }
}
```

DISCUSSION: First, a surprising number of people wrote code that explicitly managed arrays or linked lists, recreating the capabilities of existing classes/interfaces like `ArrayList` and `SimpleList` (from class). Doing so made the problem more difficult.

Deeper problems occurred when the roles of the `Deck` and the `GrabBag` were oddly mixed. Some changed their `Deck` class to store the `Card`s in a `GrabBag` at all times. This change was not only unnecessary, but also removed from the `Deck` its central property: an *ordered* collection of `Card`s. The use of an array or some kind of list as the primary container should not have been changed.

Similarly, some used two `GrabBags`, pulling items from one to put them into the other as an attempt at performing a `shuffle()`. However, taking a set of `Card`s from one unordered container and moving them to another achieves nothing. The result is not shuffled at all, since a shuffle implies an order.

3. QUESTION: Write the method ...

```
public static void permutations (int n)
```

... such that it prints every possible permutation of the integers from 0 to $n-1$.

ANSWER: One possible solution...

```
public static void permutations (int n) {
    doPerms(n, new ArrayList<Integer>());
}

private static void doPP (int n, List<Integer> sequence) {

    if (n == 0) {

        for (int i = 0; i <= sequence.size(); i += 1) {
            System.out.printf("%d\t", sequence.get(i));
        }
        System.out.println();
        return;

    }

    for (int i = 0; i <= sequence.size(); i += 1) {
        sequence.add(i, n);
        doPP(n - 1, sequence);
        sequence.remove(i);
    }

}
```

DISCUSSION: This question gave most people fits, but it also yielding some interesting answers. There were a number of people who, using no recursion, attempted a pattern of swaps and rotations of the values in an array/list. The best of these were on a useful track, but would fail to produce all possible orderings.
It is worth noting that this was a difficult problem. It was intended to be the greatest algorithmic challenge on the exam, and only a few came up with fully correct solutions. Partial credit was common.

4. QUESTION: Recall that the `SimpleList<T>` interface looked something like the following:

```
public interface SimpleList<T> {

  // Insert the value v at position i.  If not
  // 0 <= i <= length, then throw the given exception.
  // Note that i == length implies appending the list.
  public void insert (int i, T v) throws IndexOutOfBoundsException;

  // Remove and return the value a position i.  If
  // not 0 <= i < length, then throw the given exception.
  public T remove (int i) throws IndexOutOfBoundsException;

  // Return the value at position i.  If not
  // 0 <= i < length, then throw the given exception.
  public T valueAt (int i) throws IndexOutOfBoundsException;

  // Return the position at which the value v occurs in the list.
  // If v is not in the list, return -1.
  public int find (T v);

}
```

(a) Write a generic container class `SimpleStackList<T>` that implements the `SimpleList<T>` interface **using only** a `Stack<T>` to store and order the items inserted into the list. That is, implement the list operations (`insert/remove/valueAt/find`) using only the `push/pop/top` capabilities of a stack.

(b) For each of the four list operations, how many operations are required for a `SimpleStackList<T>` of length n? (Express your answers using the *big-O* notation.)

ANSWER: One approach, lengthy, but not too bad...

```
public class SimpleStackList<T> implements SimpleList<T> {

    private Stack<T> _storage;
    private Stack<t> _temp;

    public SimpleStackList () {
        _storage = new Stack<T>();
        _temp    = new Stack<T>();
    }

    public void insert (int i, T v) throws IndexOutOfBoundsException {
        checkRange(i + 1);
        walkTo(i);
        _storage.push(v);
        restore();
    }

    public T remove (int i) throws IndexOutOfBoundsException {
        checkRange(i);
        walkTo(i);
        T v = _storage.pop();
        restore();
        return v;
    }

    public T valueAt (int i) throws IndexOutOfBoundsException {
        checkRange(i);
        walkTo(i);
        T v = _storage.top();
        restore();
        return v;
    }
```

```
    public int find (T v) {
        int i = 0;
        while (_storage.size() > 0) {
            T current = _storage.top();
            if (v.equals(current)) {
                restore();
                return i;
            }
            i += 1;
            _temp.push(current);
        }
        return -1;
    }

    private void checkRange (int i) throws IndexOutOfBoundsException {
        if ((i < 0) || (_storage.size() <= i)) {
            throw new IndexOutOfBoundsException();
        }
    }

    private void walkTo (int i) {
        for (int j = 0; j < i; j += 1) {
            _temp.push(_storage.pop());
        }
    }

    private void restore () {
        while (_temp.top != null) {
            _storage.push(_temp.pop());
        }
    }

}
```

DISCUSSION: So long you had the idea to the use of a second stack in order to accumulate, in reverse order, the items that had to be popped off of the primary stack, all tended to go well enough. The key to making this answer reasonable in length was to write the small group of helper methods that moved into some position in the primary stack and then restored the stack to its original state. Many people lost a small number of points by failing to write these helpers and then making minor errors in re-creating the steps that those helpers would have performed.

Some tried simply to copy the stack into a list or array, perform the operation on that, and then copy the result back into a stack. This approach violates the clear intent of the question by trivializing the use of the stack by relying upon some other, linear container that already has the properties of a *list*. That was not an acceptable answer.