Solutions to Quiz 1 (October 19, 2015)

Problem 1 (Multiple Choice) (**14 points**).
Circle all correct answers for the following questions.

(a) Which of the following will fail to compile due to static checking? Treat each part as an independent piece of code.

A.
```
String s = null;
System.out.println(s.toLowerCase());
```

B.
```
int[] arr = new int[] { 1, 2, 3 };
arr[1] = 1;
```

C.
```
int[] arr = new int[] { 1, 2, 3 };
assert arr.length == 2;
```

D.
```
int[] arr = new int[] { 1, 2, 3 };
arr[3] = "4";
```

Solution.

A will compile successfully, but have a runtime error of NullPointerException because null is a hole in the type system, passing the compiler's static checks.

B will compile successfully, and reassign the second entry in the array, so that the array becomes 1, 1, 3.

C will compile successfully, and the assertion will be checked at runtime.

D does not compile, because a `String` cannot be assigned to an `int`.

■

(b) When implementing the `Object` contract:

A. `equals()` must be reflexive and symmetric, but not necessarily transitive

B. only mutable types should override `equals()` and `hashCode()`

C. two objects with the same `hashCode()` must be `equals()`

D. two objects that are `equals()` must have the same `hashCode()`

E. two objects that are not `equals()` must have different `hashCode()` values

Solution.

A is not correct, `equals()` must also be transitive.

B is not correct, *immutable* types should definitely override `equals()` and `hashCode()`.

C is not correct, two objects can share the same hashCode but not be equal.

D is correct. The essence of the `Object` contract.

E is not correct, the contract doesn't specify that objects that are not `equals()` must have different `hashCode()` values, only that objects that are `equals()` must have the same `hashCode()`. ■

(c) You are given the following data type:

```
/**
 * Represents the orders being processed in a factory.
 * A factory must always have at least one order in it.
 * Factories are mutable.
 */
class Factory {
    public List<Order> orders;
    // the orders in this factory are sorted from oldest to newest,
    // with no duplicates.

    /**
     * @return a list containing the orders of this factory,
     *         with no duplicates.
     */
    public List<Order> getOrders() {
        return orders;
    }

    // ... other code
}
```

Consider the following clients of the `Factory` data type:

```
/** @return newest order in the factory */
public Order client1(Factory factory) {
    Order newest = factory.getOrders().get(factory.orders.size()-1);
    return newest;
}

/** @return any order in the factory */
public Order client2(Factory factory) {
    Order anyOrder = factory.getOrders().get(0);
    return anyOrder;
}
```

Circle all of the true statements below.

A. `client1` is correct for the code and comments as shown

B. `client2` is correct for the code and comments as shown

C. `client1` depends on `Factory`'s representation

D. `client2` depends on `Factory`'s representation

E. Changing `orders` to `private` would fix the rep exposure of `Factory`

Solution.

A is correct, since the orders list contains at least one order, and the last one is the newest one.

B is correct, since the orders list contains at least one order, and `client2` can return any of them.

C is correct, because `client1` refers to the `orders` field directly. `Factory` has an exposed rep.

D is not correct because `client2` only depends on the `getOrders()` method, not any of `Factory`'s representation.

E is not correct because the `getOrders()` method still would return a reference to the `List<Order>` which could then be modified. ■

Problem 2 (Specs) (**18 points**).
Imagine you are given the following interface with a single method:

```
public interface Mode {

  /**
   * Finds one of the most frequent integers in an array.
   * @param  values   array in which at least one value occurs more than once.
   * @return          a number that appears most often in values
   */
  public int getMode(int[] values);
}
```

Along with a class that implements it:

```
public class MyMode implements Mode {

  /**
   * TODO
   */
  @Override public int getMode(int[] values){
      ...
  }
}
```

Write a spec for `MyMode.getMode()` in which the precondition and postcondition are **both different** from `Mode.getMode()`, while ensuring that `MyMode` is a well-defined spec that legally implements `Mode`.

(a) Precondition:

`@param`

(b) Postcondition:

```
@returns
```

Solution.

In order for the implementation to be legal, the new precondition must be weaker, and the new postcondition must be stronger. For example:

```
/**
 * Finds the most repeated integer out of an array of values.
 * @param  values   an array of integer values of length > 0
 * @return          the smallest mode of the array
 */
 */
```

Note that the empty array must either be excluded by the precondition or specified by the postcondition.

■

Problem 3 (Testing) (**18 points**).
For this spec:

```
/**
 * @param n  a nonnegative integer
 * @returns  the number of digits in a base-10 representation of n
 */
public int countDigits(int n);
```

Write a black box testing strategy for `countDigits` with exactly one good partition for `n` and exactly one good partition for the return value `result`. Each partition should be a list of well-formed mathematical expressions containing only numbers, variable or constant names, and equality or inequality operators ($<$, $>$, $=$, $\leq$, $\geq$).

(a) One partition for `n`:

Solution.

`n=0, n=1, 1<n<10, n=10, 10<n<MAXINT, n=MAXINT`

This partition uses four boundary values. It is a partition because it covers the entire legal space for `n`.

Note that the specification requires that `n` be nonnegative, so we don't have to (and cannot) test `n<0`.

■

(b) One partition for `result`:

Solution.

`result=1, result>1`

It isn't necessary to use MAXINT as a boundary value here, because we can never plausibly reach it.

■

Problem 4 (ADTs) (**20 points**).
Louis Reasoner has written an ADT for keeping track of relationships among strings. Unfortunately, he hasn't taken 6.005 and doesn't understand the concepts that make ADTs powerful.

```
/**
 * Represents a list of collections, where a collection is a set of strings
 * that are related for some reason, such as:
 *    - synonyms in English, e.g. {"tool", "instrument", "utensil"}
 *    - synonyms in different languages, e.g. {"tool", "outil", "instrumento"}
 * Each collection is considered fixed, so it never changes once created.
 * But when new collections are discovered, they may be added to this list.
 */
public class StringCollection {
    public final List<Set<String>> collections;

    /** Make an empty StringCollection */
    public StringCollection() {
        this.collections = new ArrayList<Set<String>>();
    }

    /** Make StringCollection from an existing StringCollection
     *  @param oldCollection  */
    public StringCollection(StringCollection oldCollection) {
        this.collections = oldCollection.collections;
    }
22
23      /** Add a new collection of strings
24       *  @param newCollection set of strings that are related to each other */
25      public void addCollection(Set<String> newCollection) {
26          this.collections.add(newCollection);
27      }
28
29      /** Get all collections known to this StringCollections object
30       *  @return the collections in this object */
31      public List<Set<String>> fetchAll() {
32          return this.collections;
33      }
34
35      /** Get all known collections that share a particular word
36       *  @param filterWord String to look for
37       *  @param result  list that receives the collections found
38       *  Adds all collections that contain filterWord to the result list. */
39      public void filter(String filterWord, List<Set<String>> result) {
40          for (Set<String> collection : this.collections) {
41              if (collection.contains(filterWord))
42                  result.add(collection);
43          }
44      }
45  }
```

(a) Classify each of the methods in StringCollection using the four types of ADT operations.

```
_____________________ StringCollection()
_____________________ StringCollection(StringCollection oldCollection)
_____________________ void addCollection(Set<String> newCollection)
_____________________ List<Set<String>> fetchAll()
_____________________ void filter(String filterWord, List<Set<String>> result)
```

Solution.

```
creator
producer
mutator
observer
observer
```

■

Unfortunately, this ADT is littered with representation exposure issues. Lend your knowledge to Louis and clean up his code!

Which lines are responsible for representation exposure? Write:

- the line number
- a one-sentence reason that the line causes rep exposure
- a one-sentence fix to it that still satisfies the spec.

There are more boxes below than you need.

(b) Line #: Reason/Fix:

Solution.

(10) makes the `collections` rep field publicly-accessible. Make it private ■

(c) Line #: Reason/Fix:

Solution.

(20) Shares a mutable list instance between the two StringCollections. Make a deep copy of the collections list, or use a rep-exposure free version of `fetchAll()`, or make a new list containing the sets wrapped in unmodifiable wrappers. ■

(d) Line #: Reason/Fix:

Solution.

(26) Sets are mutable. Keeping a reference to a passed in set means the client can modify the list which explicitly goes against the spec of the class. Make a defensive copy of the set. An unmodifiable wrapper around the set is not a good fix, because it still shares the underlying set with the client. ■

(e) Line #: Reason/Fix:

Solution.

(32) Returns a reference to collections out of the ADT, which can then be modified by adding new collections or changing existing collections, which is against the spec. The answer is to deep copy the list we want to return or allocate a new list with unmodifiable sets around the underlying sets. ■

(f) Line #: Reason/Fix:

Solution.

(42) The rep exposure here is directly adding a set to the result list. If we simply copy the set here, we don't risk the client being able to interfere with the rep of this ADT. ■

(g) Line #: Reason/Fix:

Problem 5 (Scopes) (**18 points**).
Suppose we have the following classes.

```
public class WordList {
    private List<String> wordList;
    public Frequency frequency;
    public static int maxSize;
    // other code ...
}

public class Frequency {
 9      public static int max;
10      // other code...
11
12      public Map<Integer, Set<String>> invertFrequencies(Map<String, Integer> frequencies) {
13          Set<String> words;
14          Integer i;
15          Map<Integer, Set<String>> reverseMap = new HashMap<Integer, Set<String>>();
16
17          for (String s: frequencies.keySet()) {
18              i = frequencies.get(s);
19              if (!reverseMap.containsKey(i)) {
20                  words = new Hashset<String>();
21                  words.add(s);
22                  reverseMap.put(i, words);
23              }
24              else {
25                  reverseMap.get(i).add(s);
26              }
27          }
28          return reverseMap;
29      }
30  }
```

(a) Which of these pairs of variables have the same scope of access? (Select all that apply)

A. `maxSize`, `max`

B. `max`, `words`

C. `wordList`, `frequency`

D. `frequencies`, `reverseMap`

E. `s`, `i`

Solution.

A is correct because maxSize and max are both global variables, accessible from anywhere in the program.

B is incorrect because max is a global variable, but words is a local variable inside invertFrequencies.

C is incorrect because wordList is a private instance variable accessible only to code inside WordList, but frequency is a public instance variable accessible from any code with a reference to a WordList object.

D is correct because frequencies is a parameter whose scope is the body of invertFrequencies, and reverseMap is a local variable with the same scope.

E is incorrect because the scope of i is the outermost curly-brace block of invertFrequencies, while the scope of s is just the body of the for loop.

■

(b) Two variables can be moved to minimize their scopes, without affecting any other code. Write down the variable name and the line number that its variable declaration should be moved to.

Variable: Declaration:
Variable: Declaration:

Solution. i: 18, words: 20 ■

Problem 6 (AF/RI) (**12 points**).
Consider this ADT:

```
/**
 * Represents one of the suits in a standard 52 card deck - clubs, hearts,
 * spades, or diamonds.
 */
public class CardSuit {

    private int suit;

    private static final int CLUBS = 0;
    private static final int DIAMONDS = 1;
    private static final int HEARTS = 2;
    private static final int SPADES = 3;

    public CardSuit(int suit) {
        this.suit = suit;
    }

    @Override
    public String toString() {
        switch (suit) {
            case CLUBS:    return "clubs";
            case DIAMONDS: return "diamonds";
            case HEARTS:   return "hearts";
            case SPADES:   return "spades";
            default:       assert false; // shouldn't get here
        }
    }
}
```

(a) What is the domain of the abstraction function?

Solution. int ■

(b) What is the range of the abstraction function?

Solution. The four suits in a deck - clubs, hearts, diamonds, and spades ■

(c) What is the rep invariant?

Solution. $0 <= \text{suit} <= 3$ ■