Quiz 1 (October 19, 2015)

Your name:____

Your Athena username:____________________________________

You have 50 minutes to complete this quiz. It contains 9 pages (including this page) for a total of 100 points.

The quiz is closed-book and closed-notes, but you are allowed one two-sided page of notes.

Please check your copy to make sure that it is complete before you start. Turn in all pages, together, when you finish. Before you begin, write your name on the top of every page.

Please write neatly. **No credit will be given if we cannot read what you write.**

For questions which require you to choose your answer(s) from a list, do so clearly and unambiguously by circling the letter(s) or entire answer(s). Do not use check marks, underlines, or other annotations – they will not be graded.

Good luck!

DO NOT WRITE BELOW THIS LINE

Problem	Points	Grade	Grader
1: Multiple Choice	14		
2: Specs	18		
3: Testing	18		
4: ADTs	20		
5: Scopes	18		
6: AF/RI	12		
Total	100		

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";
```

(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

(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`

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`

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`:

(b) One partition for `result`:

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.

```java
/**
 * 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;
    }

    /** Add a new collection of strings
     *  @param newCollection set of strings that are related to each other */
    public void addCollection(Set<String> newCollection) {
        this.collections.add(newCollection);
    }

    /** Get all collections known to this StringCollections object
     *  @return the collections in this object */
    public List<Set<String>> fetchAll() {
        return this.collections;
    }

    /** Get all known collections that share a particular word
     *  @param filterWord String to look for
     *  @param result  list that receives the collections found
     *  Adds all collections that contain filterWord to the result list. */
    public void filter(String filterWord, List<Set<String>> result) {
        for (Set<String> collection : this.collections) {
            if (collection.contains(filterWord))
                result.add(collection);
        }
    }
}
```

(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)
```

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:	
(c) Line #:		Reason/Fix:	
(d) Line #:		Reason/Fix:	
(e) Line #:		Reason/Fix:	
(f) Line #:		Reason/Fix:	
(g) Line #:		Reason/Fix:	

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

```
 1  public class WordList {
 2      private List<String> wordList;
 3      public Frequency frequency;
 4      public static int maxSize;
 5      // other code ...
 6  }
 7
 8  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`

(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:

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?

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

(c) What is the rep invariant?