CSE 331 Midterm Exam   11/18/13

Name ____

There are 13 questions worth a total of 100 points. Please budget your time so you get to all of the questions. Keep your answers brief and to the point.

The exam is closed book, closed notes, closed electronics, closed mouth, open mind.

Many of the questions have short solutions, even if the question is somewhat long. Don't be alarmed.

If you don't remember the exact syntax of some command or the format of a command's output, make the best attempt you can. We will make allowances when grading.

Relax, you are here to learn.

Please wait to turn the page until everyone is told to begin.

Score _________________ / 100

1.	______ / 10	8.	______ / 5
2.	______ / 16	9.	______ / 3
3.	______ / 10	10.	______ / 6
4.	______ / 10	11.	______ / 3
5.	______ / 5	12.	______ / 5
6.	______ / 14	13.	______ / 1
7.	______ / 12		

CSE 331 Midterm Exam 11/18/13

Question 1. (10 points) (assertions) Using backwards reasoning, find the weakest precondition for each sequence of statements and postcondition below. Insert appropriate assertions in each blank line. You should simplify your final answers if possible.

(a)

```
{ ____________________________________________ }

b = 3;

{ ____________________________________________ }

a = a + 1;

{ a*b > b+1 }
```

(b)

```
{ ____________________________________________ }

x = x * y;

{ ____________________________________________ }

n = n - 1;
```

{ x * y^n = b } *(y^n means y raised to the power n)*

Question 2. (16 points) Loop development. Implement the following method that returns true if a list of `String`s is sorted in non-decreasing order and false if not, and prove that your implementation is correct. The method heading is provided for you. You should use the `compareTo` method in Java code to determine the ordering of `String`s (but feel free to use ordinary notation like <= in the proof steps). You may declare any additional simple variables needed, but no additional lists or other containers. The method must not modify its argument.

You will need to provide a suitable loop invariant and other assertions, preconditions, and postconditions to prove that your code is correct.

```
   // return true if the list elements are in non-decreasing
   // order (i.e., strs.get(0) <= strs.get(1) <= ...).
   boolean isSorted(ArrayList<String> strs) {






   }
```

The next several questions concern the following partially implemented class, which implements a Map (a set of <key,value> pairs) using a pair of lists to hold the keys and values. Although the class is incomplete and the comments are not sufficient, the methods shown here do work as intended.

You can remove this page for reference as you work on the next questions.

```
public class ExamMap<K,V> {

  // instance variables
  private List<K> keys;
  private List<V> values;

  // construct a new, empty ExamMap
  public ExamMap() {
    keys = new ArrayList<K>();
    values = new ArrayList<V>();
  }

  // add <key,value> to map
  public V put(K key, V value) {
    int loc = indexOf(key);
    if (loc != -1) {
      V result = values.get(loc);
      values.set(loc, value);
      return result;
    }
    keys.add(key);
    values.add(value);
    return null;
  }

  // return index of key-value pair with matching key
  // or -1 if not found
  private int indexOf(K key) {
    for (int i = 0; i < keys.size(); i++) {
      if (key.equals(keys.get(i))) {
        return i;
      }
    }
    return -1;
  }

  // return the keys stored in this ExamMap
  public List<K> getKeys() {
    return keys;
  }
}
```

Question 3. (10 points) Give (i) a suitable class description, (ii) an *abstraction function*, and (iii) a *representation invariant* for this class. The description should be a suitable JavaDoc comment that would appear right above the "`class ExamMap<k,v>`" line at the beginning of the code. The abstraction function and representation invariant should contain the information we would expect to be included in comments right after the declarations of instance variable `keys` and `values`.

Question 4. (10 points) The `put` method adds a new <key,value> pair to the map. But it is does not have a proper specification. Below write a suitable JavaDoc comment using CSE331 conventions (@param, @requires, @modifies, etc.) to specify this method. The existing implementation must, of course, satisfy the specification you give here.

Queston 5. (5 points) Are there any potential *representation exposure* problems in the existing code? If so, what are they and how would you fix them while still providing operations for existing client code?

Question 6. (14 points) We would like to add a method to retrieve the value associated with a key. The method heading is: `public V get(K key) { ... }`. Here are four possible specifications for this method:

```
/** SPEC A
  * @requires key is not null and key is a key in this
  * @return the value associated with key
  */

/** SPEC B
  * @requires key is a key in this
  * @return the value associated with key
  * @throws NullPointerException if key is null
  */

/** SPEC C
  * @return the value associated with key if key is a key in
  *       this, or null if key is not associated with any value
  */

/** SPEC D
  * @return the value associated with key
  * @throws NullPointerException if key is null
  * @throws NoSuchElementException if key is not a key in this
  */
```

And here are some possible implementations:

```
// Implementation 1:
public V get(K key) {
  return values.get(indexOf(key));
}

// Implementation 2:
public V get(K key) {
  if (key==null) {
    throw new NullPointerException("null key passed to get");
  }
  return values.get(indexOf(key));
}

// Implementation 3:
public V get(K key) {
  if (key == null || indexOf(key) == -1) {
    return null;
  }
  return values.get(indexOf(key));
}
```

(continued next page – you may remove this page while you work if it is convenient.)

Question 6. (cont.)

```
// Implementation 4:
public V get(K key) {
  if (key == null) {
    throw new NullPointerException("null key passed to get");
  }
  if (indexOf(key) == -1) {
    throw new NoSuchElementException("key not found");
  }
  return values.get(indexOf(key));
}
```

(a) (6 points) Compare specifications. For each of the following pairs of specifications, **circle** the letter of the specification that is **stronger**. Circle "neither" if the specifications are either equivalent or incomparable, or if a specification contains an error or inconsistency.

(i)	A	B	neither
(ii)	A	C	neither
(iii)	A	D	neither
(iv)	B	C	neither
(v)	B	D	neither
(vi)	C	D	neither

(b) (8 points) Implementations and specifications. In the following table, place an X in the square if the implementation whose number is given to the left satisfies the specification whose letter is given at the top. Leave the entry blank if the implementation does not satisfy the specification or if a specification contains an error or inconsistency.

	Spec. A	Spec. B	Spec.C	Spec. D
Impl. 1				
Impl. 2				
Impl. 3				
Impl. 4				

Question 7. (12 points) Testing. We would like to test the `put` method that adds new <key,value> pairs to the ExamMap. (This is the method whose specification comment you provided in a previous question.)

(a) Describe two good **black box** (i.e., specification) tests for this method. The two tests should be from different revealing subdomains – i.e., they should not detect exactly the same set of errors. You do not need to give JUnit code – just describe the tests. You may assume that the `get` method is available if that is useful.

(i)

(ii)

(b) Describe two good **white box** (or glass box) tests for this method. As with the black box tests, the two tests should be from different revealing subdomains. Again, no JUnit code required, and you may assume the `get` method is available.

(i)

(ii)

Question 8. (5 points) Finally, we would like to add a suitable `hashCode` method to our `ExamMap` class. For this question, assume that we have defined a suitable `equals` method for `ExamMap`. Two `ExamMap`s are considered to be equal if they contain the same sets of <key,value> pairs. If we have two <key,value> pairs `<k1,v1>` and `<k2,v2>`, they are equal (i.e., the same) if `k1.equals(k2)` and `v1.equals(v2)`.

Here is our proposed implementation of `hashCode`:

```
public int hashCode() {
  int result = 0;
  for (int i = 0; i < keys.size(); i++) {
    result += 37*i*keys.get(i).hashCode() +
              31*i*values.get(i).hashCode();
  }
  return result;
}
```

Is this a correct `hashCode` implementation for `ExamMap`, given the definition of equality for `ExamMap`s described above? If so, is it a good implementation, and why? If not, what is wrong with it?

A few short answer questions…

(Meaning, please keep your answers short and to the point. A couple of sentences should be enough most of the time.)

Question 9. (3 points) One of the principles that the *Pragmatic Programmer* emphasizes is *DRY*. What does it stand for and what is the significance?

Question 10. (6 points) (a) Give one advantage of defining a type with a Java interface instead of an abstract class.

(b) Give one advantage of defining a type with an abstract class instead of a Java interface.

Question 11. (3 points) When and where should you use the `@override` annotation in a program? What is the reason for using it?

Question 12. (5 points) Circle true or false for each of the following.

true	false	The implementation of `equals()` in class `Object` is equivalent to ==.
true	false	If A is a Java subclass of B, then A.getClass().equals(B.getClass()) is true.
true	false	If A is a Java subclass of B, then `A instanceof B` is true.
true	false	The method `isEmpty()` in the Java Collections interface is an "observer" method.
true	false	The method `clear()` in the Java Collections interface is a "creator" method.

Question 13. (1 point – all honest answers receive the point) What is the one question (if any) that you really thought would be on this test that we forgot to include??