CSE 331 Midterm Exam Sample Solution 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)

{ __(a+1)*3 > 3+1__ } **=> { 3*a + 3 > 4 } => { 3*a > 1 }**
(which implies a>=1 assuming integer arithmetic, but it was fine to leave the final assertion as {3a>1}.)

```
b = 3;
```

{ __(a+1) * b > b+1__ }

```
a = a + 1;
```

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

(b)

{ __(x * y) * y^{n-1} = b__ } **=> { x * y^n = b }**

```
x = x * y;
```

{ __x * y^{n-1} = b__ }

```
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.

Note: The assertions use “strs[k]” instead of “strs.get(k)” to save a little space.

```
// return true if the list elements are in non-decreasing
// order (i.e., strs.get(0) <= strs.get(1) <= ...).
boolean isSorted(ArrayList<String> strs) {
   if (strs.size() == 0) {     // simplify handling of empty
      return true;             // list case by checking here
   }
   int k = 1;
   { inv: strs[0] <= strs[1] <= ... <= strs[k-1] }
   while (k != strs.size())
      { inv && k < strs.size() }
      if (strs.get(k-1).compareTo(strs.get(k)) <= 0) {
         { inv & strs[k-1]<=strs[k] }
         k = k+1;
         { inv }
      } else {
         { inv & strs[k-1] > strs[k] => list is not sorted }
         return false;
      }
   }
   { inv && !(k!=strs.size() } =>
   { inv && k==strs.size() } =>
   { strs[0] <= strs[1] <= strs[strs.size()-1] }
   return true;
}
```

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`.

Description:
An ExamMap is an unordered collection of <key, value> pairs that implements a map from keys to values.

Abstraction Function:
An ExamMap is a set of <key,value> pairs. For 0 <= i < keys.size(), keys.get(i) is the key of a pair and values.get(i) is the associated value.

Representation Invariant:
keys != null && values != null && keys.size() == values.size() &&
for 0 <= i < keys.size(), keys.get(i) and values.get(i) form a <key,value> pair &&
for 0 <= i, j < keys.size(), keys(i).equals(keys(j)) is false unless i=j.

Notes: A realistic representation invariant would probably require that the elements of `keys` not be `null`, and probably would also require that the elements in `values` not be `null` either so functions like `equals` and `hashCode` could reference those items. But the code on the previous page works without these restrictions so we did not require them in the representation invariant.

We also didn't deduct points if the representation invariant did not mention that instance variables `keys` and `values` should not be `null`. That should be included for completeness but the main issues we were looking for were the relationships between the lists and the <key,value> pairs contained in them.

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.

```
/**
 *  Add a <key,value> pair to this ExamMap, replacing the
 *  associated value if key is already present in the map
 *
 *  @param key The key of the <key,value> pair to be added
 *
 *  @param value The value of the <key,value> pair to be
 *               added.
 *
 *  @requires key != null   (* - see remark below)
 *
 *  @modifies this
 *
 *  @effects <key, value> is added to this.  If key is
 *           already present, replace the associated value.
 *
 *  @return the old value associated with key if key was
 *          already present in this, otherwise null if
 *          inserting a new key in the map.
 */
```

(*The code on the previous page for `indexOf`, which is called by `put`, will fail if given a `null` key. This is a subtle point so it was only a minor deduction if it was missed.)

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?

Yes. Method `getKeys` returns a reference to the `keys` list. Client code could use that reference to alter `keys` directly without going through the proper interfaces.

One possible solution is to return a clone of `keys` to the caller. Another would be to wrap `keys` in an `unmodifiableList` to give the client a view of the list but without the ability to change it.

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	X			
Impl. 2	X	X		
Impl. 3	X		X	
Impl. 4	X	X		X

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.

There are obviously many possibilities. Here are a few:

- **Create an empty map. Verify that `put` of a new <key,value> pair returns `null`. Verify that `get` with that key returns the associated value.**
- **Create a map populated with several <key,value> pairs. Add a <key,value> pair whose key is different than any previous key and verify that `put` returns `null`. Verify that `get` with that key returns the associated value. Verify that `get` continues to return the correct values for all of the other keys.**
- **As with the previous test, create a map populated with several <key,value> pairs. Add a <key,value> pair whose key is the same as one of the previous keys. Verify that `put` returns the old value associated with the key. Verify that `get` with that key now returns the new value. Verify that `get` with the remaining keys returns correct values.**

For full credit the answers had to give specific details of the test setup and how the result would be verified.

(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.

As in (a) there are many possibilities. Here are a few:

- **Create an empty map. Verify that `keys` and `values` both refer to `ArrayLists` that contain no elements.**
- **Add a <key,value> pair to an empty map. Verify that the key and value are the single elements in the `keys` and `values` lists respectively.**
- **Create a map with several <key,value> pairs. Add a new <key,value> pair whose key differs from any previous key. Verify that the key and value are at the end of the `keys` and `values` lists and that the lists have the same lengths.**
- **Create a map with several <key,value> pairs. Add a new <key,value> pair whose key matches an existing key in the middle of the `keys` list. Verify that the corresponding element in `values` has been updated.**
- **Same test as the previous one, but replace the value whose <key,value> pair appears at the beginning of the two lists.**

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?

No, this will not work. It will return different `hashCodes` for two `ExamMaps` that contain the same set of <key,value> pairs stored in different orders in the lists.

Note: although this particular implementation stores keys in the list in the order they were added, that does not imply that there is a unique representation for a map containing the same <key,value> pairs. For example, suppose we create two maps, add <k1,v1> and <k2,v2> to the first one in that order, then add <k2,v2> and <k1,v1> to the second map in the other order. These two maps represent the same abstract value, they should be treated as equal, and they should have the same `hashCodes`.

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?

Don't Repeat Yourself. The quote from the book is "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."

Answers that captured the essence of this received full credit. It wasn't necessary to produce the exact quote from memory.

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

Here are two:

- **Multiple classes can implement the same interface so they have a common type (behavior) even if they are not related by inheritance.**
- **A class can have multiple interfaces (types) in addition to the single chain of types that it has due to inheritance.**

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

The main advantage is that an abstract class can contain implementations of some or all of the methods of the type. Classes extending the abstract class can inherit these implementations.

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

@override is written before a method declaration to indicate that this method is intended to override a method that would otherwise be inherited from a superclass. It allows the compiler to report a problem if the method doesn't actually override any inherited method because, for example, it has a different number of parameters or the parameter types don't match.

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??

???