Name:____

CSE331 Spring 2015, Midterm Examination May 6, 2015

Please do not turn the page until 10:30.

Rules:

- The exam is closed-book, closed-note, etc.
- **Please stop promptly at 11:20.**
- There are **100 points**, distributed **unevenly** among **9** questions (all with multiple parts):

Question	Max	Earned
1	17	
2	7	
3	22	
4	12	
5	10	
6	10	
7	8	
8	8	
9	6	

Advice:

- Read questions carefully. Understand a question before you start writing.
- **Write down thoughts and intermediate steps so you can get partial credit. But clearly indicate what is your final answer.**
- The questions are not necessarily in order of difficulty. **Skip around.** Make sure you get to all the questions.
- If you have questions, ask.
- Relax. You are here to learn.

Name:____

1. (**17** points) Consider this code with the indicated initial precondition where `arr` is an `int[]`:

```
{arr != null /\ arr.length > 1}
int p = arr[0];
int i = 1;
while(i != arr.length - 1) {
    int t = arr[i];
    arr[i] = (p + t + arr[i+1]) / 3;
    p = t;
    i = i+1;
}
```

(a) In 1–2 English sentences, what is the post-condition for this code? Discuss only the final contents of `arr` (not other variables), referring to the original contens of `arr` as needed. Note you need to describe some of the array contents differently from the rest.

(b) Now give a formal mathematical description of the post-condition you described in English in part (a). Use this notation as needed:

- Use `arr[i]pre` to mean the contents of `arr[i]` before any of the code executes.
- Use `x..y` to mean the range of numbers from `x` to `y`, *including* `x` and *excluding* `y`. (If $y \leq x$, then this means the empty range.)

(c) Now give a correct loop invariant for the loop in the program. Make sure your loop invariant holds and is sufficient to prove the post-condition. To save time, we are *not* asking you to write out other assertions, but you will likely need to think through them and try to write some of them to get this problem right. Hint: You need several facts "and-ed together" (probably 4 or 5 depending how you structure things), with partial credit for weaker invariants that get some of the right information.

Solution:

(a) The first and last elements of the array are unchanged. For all other elements `i`, the value in `arr[i]` is changed to the average of the original values of `arr[i-1]`, `arr[i]`, and `arr[i+1]`.

(b)
```
arr[0] = arr[0]pre /\ arr[arr.length-1] = arr[arr.length-1]pre
  /\ forall j in 1..arr.length-1 arr[j] = (arr[j-1]pre + arr[j]pre + arr[j+1]pre)/3
```

(c)
```
arr[0] = arr[0]pre
/\ forall j in 1..i arr[j] = (arr[j-1]pre + arr[j]pre + arr[j+1]pre)/3
/\ p=arr[i-1]pre
/\ forall j in i..arr.length arr[j] = arr[j]pre
```

You can also include `arr[arr.length-1] = arr[arr.length-1]pre`, though technically this is included in the last line above (though it wouldn't be if you made the range `i..arr.length-1`).

Name:____

2. (**7** points)

   (a) What is wrong with this method specification?

   ```
   /**
    * @requires p != null and x > 0
    * @returns square root of (max(p.y, x))
    * @modifies p
    * @effects decrements p.y
    * @throws NullPointerException if p == null
    */
   double m(SomeClassWithPublicFieldY p, int x) {...}
   ```

   (b) What is wrong with this method specification?

   ```
   /**
    * @effects adds 7 to every element in arr
    */
   void m(int[] arr) {...}
   ```

Solution:

(a) It specifies what happens (namely an exception is thrown) in (some) cases where the requires clause is not met.

(b) It specifies an object (the array referred to by `arr`) is modified in some way, but the object is not in the `modifies` clause.

Name:____

3. (**22** points) This problem considers an ADT for a *non-decreasing finite sequence of ints*. An abstract value is a sequence of integers where each number is no less than the preceding one. An example would be -1, -1, 17, 19, 34, 34, 34, 40. It is a mutable ADT where numbers can be added only to the large end of the sequence.

   Our concrete representation uses an `ArrayList<IntPair>` where for each `IntPair`, `count` indicates how many times `num` is in the sequence. This representation can be particularly efficient when a sequence has numbers repeated many times.

   Our representation invariant includes the following:

   - The `ArrayList` is *sorted* by `num`: The `num` field of any element is greater than the `num` field of all preceding elements.
   - The `ArrayList` is as short as possible. In particular, no `num` field is repeated (instead there is just a higher `count` value for that number).

   Here is a partial implementation:

```
class IntPair {
    public int num;
    public int count;
    public IntPair(int n, int c) { num=n; count=c; }
}

public class Sequence {
    private ArrayList<IntPair> arr;

    public Sequence(int i) {
        arr = new ArrayList<IntPair>();
        arr.add(new IntPair(i,1));
    }

    public String toString() { // for simplicity has extra " " at end
        // algorithm is inefficient but simple
        String ans = "";
        for(IntPair p : arr) {
            for(int c = p.count; c > 0; c--)
                ans += p.num + " ";
        }
        return ans;
    }

    ... more methods ...
}
```

   See the next page for questions.

Name:____

Parts (b), (c), and (e) ask you to write code. You do *not* need to write specifications or comments.

(a) Can two instances of the ADT with different array-list contents (i.e., there is one or more `i` such that the objects have a different value for `arr.get(i).num` or `arr.get(i).count`) represent the same abstract value? If yes, give an example. If no, no explanation needed.

(b) Write a `checkRep` method for this implementation. Note the "short as possible" invariant may require checking some properties not explicitly mentioned on the previous page.

(c) Write a public observer method `mostCommon` that returns an `IntPair` containing the number that appears most often in the sequence and the number of times that number appears. Resolve ties however you want.

(d) Does your implementation of `mostCommon` suffer from representation exposure? If so, identify the exposure. If not, identify what you did to avoid it.

(e) Write a public mutator method `addToEnd` that adds a number to the sequence. It should throw the checked exception `NotLargeEnoughException` if the number is too small to be added. Assume the class for this exception is already defined.

Solution:

(a) No

(b)
```
    private void checkRep() {
        assert(arr != null && arr.size > 0); // we didn't take off for omitting these
        IntPair prev = arr.get(0);
        assert(prev != null && prev.count > 0);
        for(int i=1; i < arr.size(); i++) {
            IntPair p = arr.get(i);
            assert(p != null && p.count > 0 && p.num > prev.num);
            prev = p;
        }
    }
```

(c)
```
    public IntPair mostCommon() {
        IntPair ans = arr.get(0);
        for(IntPair p : arr)
            if(p.count > ans.count)
                ans = p;
        // has rep-exposure: return ans;
        return new IntPair(ans.num,ans.count); // no rep-exposure
    }
```

(d) No, it returns a copy of the `IntPair` to avoid creating an alias to mutable internal data. (Or yes, it returns an alias to an `IntPair` used in the representation.)

(e)
```
    public void addToEnd(int i) throws NotLargeEnoughException {
        IntPair lastPair = arr.get(arr.size()-1);
        if(lastPair.num > i)
            throw new NotLargeEnoughException();
        if(lastPair.num == i)
            lastPair.count++;
        else
            arr.add(new IntPair(i,1));
    }
```

Name:____

4. (**12** points) This problem continues the previous problem. We consider implementing the same ADT, still with an `ArrayList<IntPair>`, but with a different representation invariant. We no longer maintain that the `ArrayList` is as short as possible: We allow (adjacent) array elements to have the same `num` value. We still require it to be sorted.

   (a) Can two instances of the ADT with different array-list contents (i.e., there is one or more `i` such that the objects have a different value for `arr.get(i).num` or `arr.get(i).count`) represent the same abstract value? If yes, give an example. If no, no explanation needed.

   (b) Is the new representation invariant *weaker*, *stronger*, or *incomparable* to the one in the previous problem? No explanation needed.

   (c) Given the new representation invariant, does the implementation of `toString` need to change? No explanation needed.

   (d) Given the new representation invariant, does the implementation of `mostCommon` need to change? No explanation needed.

   (e) Given the new representation invariant, does the implementation of `addToEnd` need to change? No explanation needed.

   Solution:

   (a) Yes, for example one containing a single IntPair with num 0 and count 2 versus one containing two IntPairs with num 0 each with count 1.

   (b) weaker

   (c) no, it still produces the same string representing the abstract value

   (d) yes, it would now need to check for more than one element containing the same num and sum all the counts for such elements

   (e) We ended up giving both "yes" and "no" credit for this problem, as well as "it depends" answers based on this ambiguity: In problem 3, the "shortest as possible" aspect of the rep-invariant implies that there are no counts of 0. In Problem 4, it is not clear whether the count for an `IntPair` can be 0. If not, then the answer is "no," else the answer is "yes."

   (By the way, in the "no" case, there is still now a simpler implementation that always adds a new IntPair but this approach uses more space.)

Name:____

5. (**10** points)

   (a) For any ADT, we recommend writing a Java method that checks the representation invariant. Give two distinct reasons why we recommend writing this code (rather than only describing the representation invariant in comments).

   (b) Why should the Java method for checking a representation invariant be private?

   (c) For any ADT, there should be an abstraction function, but we do not recommend writing a Java method for it. Why not?

   Solution:

   (a) First, code is more precise and complete than comments. Second, it is invaluable for debugging, especially if it is called at the beginning and end of each method where the rep invariant should hold.

   (b) The representation invariant is about the implementation and clients should not know anything about the implementation.

   (c) The abstraction function produces an abstract value, which cannot be represented in Java directly (only concrete values can), so such code cannot be written.

Name:____

6. (**10** points) Consider these two class definitions:

```
public class Fruit {
    public String color;
    public double weight;

    public Fruit(String c, double w) { color=c; weight=w; }

    public boolean equals(Object o) {
        if(!(o instanceof Fruit))
            return false;
        Fruit f = (Fruit) o;
        return this.color.equals(f.color); // only compare colors
    }
    // other methods...
}

public class Apple extends Fruit {
    public boolean isShiny;

    public Apple(String c, double w, boolean s) { super(c,w); isShiny=s; }

    public boolean equals(Object o) {
        if(!(o instanceof Fruit))
            return false;
        boolean ans = true;
        ans = ans && super.equals(o);
        ans = ans && ((Fruit) o).weight == this.weight;
        if(o instanceof Apple) {
            Apple a = (Apple) o;
            ans = ans && this.isShiny == a.isShiny;
        }
        return ans;
    }
    // other methods...
}
```

(a) Do these `equals` methods provide the *reflexivity* property of Java's equals contract? If yes, just say so. If not, provide an example violating reflexivity.

(b) Do these `equals` methods provide the *symmetry* property of Java's equals contract? If yes, just say so. If not, provide an example violating symmetry.

(c) Do these `equals` methods provide the *transitivity* property of Java's equals contract? If yes, just say so. If not, provide an example violating transitivity.

There is room on the next page for your answsers, if needed.

Name:____

Room for answers from previous page.

Solution:

(a) Yes

(b) No, for example:

```
Fruit f = new Fruit("red",0.0);
Apple a = new Apple("red",1.0,true);
f.equals(a); // true, same color
a.equals(f); // false
```

(This is a little tricky: normally we have symmetry in this sort of situation, but `Fruit` did not compare weights and `Apple` does even when comparing with a `Fruit`.)

(c) No, for example (similar to example in class):

```
Apple a1 = new Apple("Green", 2.7, false)
Fruit f = new Fruit("Green", 2.7)
Apple a2 = new Apple("Green", 2.7, true)
a1.equals(f) // true
f.equals(a2) // true
a1.equals(a2) // false, shiny values are not the same!
```

Name:____

7. (**8** points) Consider this method:

```
int m(boolean a, boolean b, boolean c, boolean d) {
    int ans = 1;
    if (a) {
        ans  = 2;
    } else if (b) {
        ans = 3;
    } else if (c) {
        if(d) {
            ans = 4;
        }
    }
    return ans;
}
```

No explanations are required below, but explanations will be needed for partial credit for wrong answers.

(a) What number of tests do you need to *exhaustively test* this method?

(b) What is the minimum number of tests needed to achieve full *statement coverage* for this method?

(c) What is the minimum number of tests needed to achieve full *branch coverage* for this method?

(d) Which of the questions above can be answered while using only a black-box testing methodology?

Solution:

(a) 16

(b) 3 (a true; a false and b true; a false, b false, c true, and d true)

(c) 5 (a true; a false and b true; a false, b false, c true, d true; a false, b false, c true, d false; a false, b false, c false)

(d) only (a)

Name:____

8. (**8** points)

Suppose `FooException` is a checked exception. Consider this code:

```
/**
 * @throws FooException if x > 0
 */
void m(int x) throws FooException {
    System.out.println(x);
}

void n() {
    m(17);
}
```

Does this code typecheck (assuming it is in some class)? If so, indicate what happens when `n` is called. If not, describe two changes where either one would cause the program to typecheck.

Solution:
No, it does not typecheck. Here are three fixes, any two get full credit:

- Add `throws FooException` to the signature for `n`.
- Put the call `m(17)` in a try/catch statement that catches `FooException`
- Remove `throws FooException` from the signature for `m`. (We can do this because the method does not actually throw an exception.)

Name:____

9. (**6** points) Short answer related to assigned readings

   (a) Yes or no: Did the class readings recommend creating strong-as-possible specifications for all classes and methods before beginning implementation?

   (b) If a subclass overrides a method defined in a superclass, how should the specification of the subclass method be related to the specification of the superclass method? (Circle one.)

   - The specification for the subclass method should be stronger (or at least no weaker).
   - The specification for the subclass method should be weaker (or at least no stronger).
   - The specification for the subclass method should be either weaker or stronger.
   - The specification for the subclass method should be neither weaker nor stronger.

   (c) What is the return type of the `compareTo` method in the `Comparable` interface?

Solution:

(a) No

(b) The specification for the subclass method should be stronger (or at least no weaker).

(c) int