CSE 331 Midterm Exam **Sample Solution** 2/18/15

Question 1. (10 points) (Forward reasoning) Using forward reasoning, write an assertion in each blank space indicating what is known about the program state at that point, given the precondition and the previously executed statements. Be as specific as possible, but be sure to retain all relevant information.

(a)

```
{ x > 0 & y > x }

y = 17;

{ x > 0 & y = 17 }

x = x * 2;

{ x > 0 & y = 17 }
```

(b)

```
{ |x| = 42 }

x = x / 2;

{ |x| = 21 } => { x = -21 | x = 21 }

x = x + 3;

{ x = -18 | x = 24 }
```

Question 2. (12 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) (5 points)

```
{ y+3 + 1 > 2*y } => { y+4 > 2*y } => { 4 > y }
x = y+3;
{ x+1 > 2*y }
w = x + 1;
{w > 2*y}
```

(b) (7 points)

```
{ x > 90 }
if (x > 0) {
   { x + 10 > 100 } => { x > 90 }
   x = x + 10;
   { x > 100 }
} else {
   { x - 20 > 100 } => { x > 120 }
   x = x - 20;
   { x > 100 }
}
{ x > 100 }
```

Note that the else branch would only be executed if x <= 0, but in that case it would be impossible to satisfy the precondition needed to ensure x > 100 after execution of the assignment x=x-20.

Question 3. (16 points) Loops. Given a positive integer *x*, the following method supposedly returns the integer value *n* such that $2^{n-1} < x \le 2^n$. Prove that the code works as specified. You need to figure out an appropriate loop invariant and then annotate the code with the invariant and other assertions needed to prove that it works.

```
// return positive integer n such that 2^(n-1) < x <= 2^n
// pre: x > 1
  static int pwr2(int x) {
    int n = 0;
    int p = 1;

    { inv: p = 2^n & x > 2^(n-1) }
    while (x > p) {

      { inv & x > 2^n }
      n = n + 1;

      { p = 2^(n-1) & x > 2^(n-1) }
      p = p * 2;

      { inv: p = 2^n & x > 2^(n-1) }
    }

   { inv & x <= p } => { p = 2^n & x > 2^(n-1) & x <= 2^n }
                   => { 2^(n-1) < x <= 2^n }
   return n;
  }
```

Question 4. (8 points) Here is the header for a method that computes a student's overall score and adds that information to a gradebook data structure.

```
void addScore(String name, List<Double> scores, Map<String,Double> gradeBook);
```

(a) (3 points) Here are two possible specifications for this method:

X @requires name != null and scores != null and gradeBook != null
@modifies gradeBook
@effects add a mapping <name, overallScore> to gradeBook

Y @requires name != null and scores != null
@modifies gradeBook
@effects add a mapping <name, overallScore> to gradeBook
@throws IllegalArgumentException if gradeBook is null

Which specification is stronger than the other? (circle) *X* [✓ *Y*] neither

(b) (3 points) Here is one possible implementation of this method:

```
if (name == null || scores == null || gradeBook == null)
   throw new IllegalArgumentException();
double grade = 0.0;
for (double s : scores)  grade += s;
if (scores.size() > 0) grade /= scores.size();
gradeBook.put(name,grade);
```

Which specification(s) does this implementation satisfy? (circle)

X *Y* [✓ both] neither

(c) (2 points) This method was designed by one of the summer interns. Is this a well-designed method or not? (i.e., think about guidelines for how to decide what methods should do, etc.) Briefly justify your answer – if it's great, say why, if it needs to be redesigned or replaced by something else, what should change and why?

It is not a particularly good design because the method is doing two things: calculating an average score and storing that information in a particular data structure. It would be better to split off the calculation into a separate method.

The next few questions concern the following class, which was written by someone who never had the advantage of learning how to specify things properly as in CSE 331. You can remove this page for reference while answering the following questions.

```
/** A FiniteStringList contains an ordered list of strings but has a finite capacity. */
public class FiniteStringList {
  private String[] strings;      // strings in this list
  private int numStrings;        // number of strings in this list

  // constructor
  public FiniteStringList(int capacity) {
    assert capacity >= 0;
    strings = new String[capacity];
    numStrings = 0;
  }

  // return number of strings currently stored in this list
  public int size() { return numStrings; }

  // add new string to the end of the list if room and return true if successful
  public boolean add(String newString) {
    assert newString != null : "cannot add null to FiniteStringList";
    if (numStrings == strings.length)
      return false;
    strings[numStrings] = newString;
    numStrings++;
    return true;
  }

  // return string in location k from the list
  public String get(int k) {
    if (k < 0 || k >= numStrings)
      throw new IndexOutOfBoundsException();
    return strings[k];
  }

  // replace string at location k with newString and return old value that was replaced
  public String set(int k, String newString) {
    assert newString != null : "cannot add null to FiniteStringList";
    if (k < 0 || k >= numStrings)
      throw new IndexOutOfBoundsException();
    String oldval = strings[k];
    strings[k] = newString;
    return oldval;
  }
}
```

Question 5. (10 points) This class lacks a proper abstract specification of the class (i.e., the specification comment that appears above the start of the class definition itself), and it does not give a representation invariant (RI) or abstraction function (AF). In the space below write a proper abstract specification, representation invariant, and abstraction function for this class. (Don't be alarmed if these are fairly simple. You may need to look at the code on the previous page to figure out the most appropriate specifications.)

```
/** (abstract class description here…)
*
*
* A FiniteStringList stores an ordered sequence of Strings s0, s1, …, sn where s0
* is the first string added to the list (or a value stored to replace that first string),
* and sn is the last string added to the list (or a value stored to replace it).
* The list has a fixed, finite capacity that is specified when it is created.
*
*
*/
public class FiniteStringList {

  // rep
  String[] strings;       // strings in this list
  int numStrings;         // number of strings in this list

// Representation invariant (RI):
//
//  numStrings >= 0 & numStrings <= strings.length &
//  strings[0..numStrings-1] does not contain null
//
//
// Abstraction Function (AF):
//
// The Strings s0, s1, …, sn are stored in strings[0], strings[1], …,
// strings[numStrings-1] in that order
//
```

Question 6. (12 points) Specifications. None of the methods in this class have a proper CSE 331-style specification. Complete the JavaDoc comments for the *set* method below to provide the most suitable specification, based on what is written in the code. Leave any unneeded fields empty. There is space at the beginning (before @param) to write the summary description of the method that should begin every JavaDoc specification. You may have to use your best judgment based on the implementation to decide how to specify some details.

```
/**
 * Replace the kth element of this with a new string and return the old value
 * stored at that location
 *
 * @param k location in this to be updated/replaced
 *
 * @param newString new value to store in location k
 *
 * @requires newString != null
 *
 * @modifies this
 *
 * @effects newString stored in position k of this
 *
 * @throws IndexOutOfBoundsException if k < 0 or k >= this.size() *
 *
 * @returns String previously stored at location k of this
 *
 */
public String set(int k, String newString) {
  assert newString != null : "cannot add null to FiniteStringList";
  if (k < 0 || k >= numStrings)
     throw new IndexOutOfBoundsException();
  String oldval = strings[k];
  strings[k] = newString;
  return oldval;
}
```

***Answers that include 0 <= k < size() as a precondition (@requires) instead of in the @throws clause received full credit. From examining the code it appears that this is being handled differently than the test for newString != null, which is clearly a precondition, but it is reasonable to throw a specific exception for a precondition violation.**

Specifications that mentioned the numStrings variable lost a small amount of credit unless numStrings was used in the abstract specification of the class as a name for the number of strings currently stored in the FiniteStringList

Question 7. (12 points) Testing. Here is the specification for a function that tests for divisibility by 3.

```
/** Test for divisibility by 3
  * @param n number to check
  * @requires n >= 0
  * @return true if n is divisible by 3
  */
public static boolean divisibleBy3(int n) { … }
```

(a) (9 points) Describe three separate black-box (specification) tests for this method. For full credit each one should test separate subdomains of the input. For each test give the input value of n and the expected result. You do not need to write junit or other Java code and you don't need to explain why you picked the tests you did.

(i) **input: n = 0, output: true**

(ii) **input: n = 1, output: false**

(iii) **input: n = 3, output: true**

There are, of course, many other possibilities. Answers received full credit if they were reasonable and not obviously testing the same subdomains repeatedly.

(b) (3 points) Here is an implementation of this method that contains a silly bug:

```
public static boolean divisibleBy3(int n) {
  if (n == 331) return true;
  return (n % 3 == 0);
}
```

What are the *revealing subdomains* (i.e., input domains) that would reveal the existence of this particular bug in the code?

{ 331 } and { all integers other than 331 }

Question 8. (14 points) Equality and things. Suppose we have a class representing movies, storing their title, year released, and length:

```
public class Movie {
  String title;       // movie title
  int    year;        // year released
  int    minutes;     // length of the movie
  …
}
```

(a) (8 points) Write a proper equals method for this class. Two Movie objects are considered to be equal if they have the same title and same year released.

```
  @Override
  public boolean equals(Object o) {
    if (! (o instanceof Movie)) {
        return false;
    }
    Movie m = (Movie) o;
    return this.title.equals(m.title) && this.year == m.year;
  }
```

Now, here are four possible implementations of hashCode for this class.

1. public int hashCode() { return 331; }
2. public int hashCode() { return title.hashCode(); }
3. public int hashCode() { return title.hashCode() + minutes; }
4. public int hashCode() { return title.hashCode() + year; }

(b) (4 points) Which of these four implementations are a correct implementation of hashCode, i.e, they obey the contract for hashCode. List the numbers of the correct ones.

1, 2, 4

(c) (2 points) Which of the four implementations of hashCode is likely to be the best one? Give a one- or two-sentence reason for your answer.

#4 is likely best since it includes all of the data involved in the equality test in the hashCode computation, which means that movies that are not considered equal are more likely to have distinct hashCodes.

Question 9. (6 points) Suppose we have a class that stores a list of some sort.

```
public class Things {
  List<String> things;
  …
}
```

We want to add a method to this class to allow clients to retrieve the current contents of the list. Here is an implementation that is clearly a bad idea because of representation exposure problems:

```
  public List<String> getThings() { return things; }
```

Here are two implementations that try to avoid the representation exposure problem:

1. `public List<String> getThings() { return new ArrayList<String>(things); }`

2. `public List<String> getThings() { return Collections.unmodifiableList(things); }`

(a) (3 points) Give one reason why we might prefer version #1 compared to #2:

Returns a new copy of the data that will not change over time. Solution #2 returns a read-only view of the list, but the client code might see different values in that list if it reads the list repeatedly while it is being changed by the Things implementation.

We might also prefer this if the client wants a mutable copy of the data, although that is a secondary reason compared to getting a fixed snapshot of all of the data at one moment in time.

(b) (3 points) Give one reason why we might prefer version #2 compared to #1.

Faster/cheaper. It returns a view of the original data without creating a copy of it.