CSE 331 Midterm Exam 5/9/14

Name _____

There are 7 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
2. _____ / 14
3. _____ / 22
4. _____ / 10
5. _____ / 18
6. _____ / 8
7. _____ / 18

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.

(a)
```
{ x < 0 & y > 0 }
y = 2;
{ ____________________________________________ }
x = x + y;
{ ____________________________________________ }
```

(b)
```
{ |x| > 2 }
x = x * 2;
{ ____________________________________________ }
x = x - 1;
{ ____________________________________________ }
```

Question 2. (14 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.

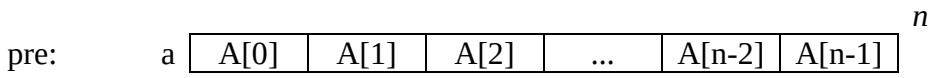
(a) (5 points)

```
{ _____________________________________ }
x = y - 2;

{ _____________________________________ }
w = x + 1;
{|w| = 10}
```

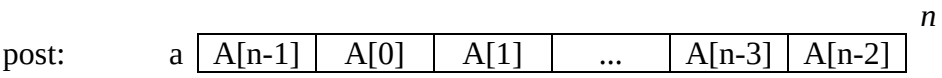
(b) (9 points)

```
{ _____________________________________ }
if (x > 4) {

   { _____________________________________ }
   x = x - 3;

   { _____________________________________ }
} else if (x < -4) {

   { _____________________________________ }
   x = x + 3;

   { _____________________________________ }
} else {

   { _____________________________________ }
   x = x + 1;

   { _____________________________________ }
}

{ x > 0 }
```

Question 3. (22 points) Loop development. For this question, implement and prove correct a method `circularShiftRight` that will rotate the elements of an array one position to the right, with the rightmost element moving to the left end. More precisely, we want to shift *n* elements of array *a*. Its initial condition (i.e., precondition) can be described by the following picture:



After the method executes, the array should be rearranged as follows:



We are using the notation A[*i*] to refer to the original value stored in a[*i*] rather than using $a_{pre}[i]$, which is more tedious to write. Feel free to use this upper-case A[i] notation to reference original values in your answer, although you can write $a_{pre}[i]$ if you wish. Just be sure it is clear what you are doing.

For full credit, your code and proof should rearrange the array in $O(n)$ time, using a single pass through the array. You must rearrange the array elements with simple assignment statements. You may use a small number of additional scalar variables in your solution, but you may not use additional data structures such as another array to hold a copy of the values. (You may, of course, use a few simple temporary variables to hold some individual array elements.) For simplicity we assume the array elements have type `int`. You may also assume that the array has at least *n* elements, but you may not make any other assumptions about how large it is. You may not call additional methods or use recursion in your solution.

(a) (4 points) Give a suitable *loop invariant* for the loop that shifts the array elements. You may draw a picture as above or use notation like a[0..k] to describe array sections.

(continued next page)

Question 3. (cont.) (b) (18 points) Implement method `circularShiftRight` below and prove that your implementation is correct using appropriate assertions, preconditions, post conditions, and invariants. Your proof should be precise and should not omit essential details or expect the reader to supply missing steps, although you may omit truly trivial assertions between consecutive statements if appropriate.

```
// Rearrange a[0..n-1] by shifting the elements
// circularly to the right. The element shifted off the
// right end from position a[n-1] should move to a[0].
void circularShiftRight(int[] a, int n) {









}
```

The next several questions concern the following partially implemented class, which stores a list of items in a restaurant menu and their prices. Menu items are `Strings` like "pizza" or "tofu". Prices are integers giving the price in pennies (for example, a price of 249 represents $2.49). Prices are stored as java `Integer` values in a `HashMap`, although all method parameters and results use ordinary Java `int`s.

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

```
public class Menu {
  // menu data (instance variable)
  private HashMap<String, Integer> items;

  /** construct empty Menu */
  public Menu() {
    items = new HashMap<String, Integer>();
  }

  /** store item in menu with given price */
  public void addItem(String name, int price) {
    items.put(name, price);
  }

  /** Return true if item is included in this menu */
  public boolean contains(String item) {
    return items.get(item) != null;
  }

  /** Return the price of the named item */
  public int getPrice(String item) {
    ... // implementation omitted
  }

  /** Return the total price of all of the items in an order (a
   *  list of item names), assuming that all of the items in
   *  the order list actually appear in the menu. */
  public int getOrderPrice(List<String> order) {
    ... // implementation omitted
  }
}
```

(additional reference information on the next page)

Reference information about maps

If `m` is variable of type `HashMap<K,V>`, where K is the key type and V is the value type, the following methods are available.

`m.containsKey(k)`	return true if k is a key in map m
`m.containsValue(v)`	return true if one or more keys in m map to the value v
`m.get(k)`	return value associated with k, or null if k is not a key in m
`m.isEmpty()`	return true if m contains no key-value mappings
`m.keySet()`	return a Set<K> view of the keys in m
`m.put(k,v)`	store value v with key k in m; return the previous value associated with k or null if k was not a key in m
`m.remove(k)`	remove any key-value mapping with key k from m
`m.size()`	return the number of key-value mappings in m
`m.values()`	return a Collection<V> view of the values in m

Useful facts about ints and Integers

Suppose we have

```
Integer num = new Integer(1);
int n;
```

Then the assignment `n=num` will store the `int` value 1 in `n`, `num+1` will evaluate to 2, and if a method is supposed to return an `int` value, it can return the `Integer` `num`, and the value 1 will be returned (i.e., an `Integer` object is converted to an `int` as needed).

However if we have `Integer num = null;` (which is legal) then the assignment `n=num` and the expression `num+1` will throw a `NullPointerException` since the `null` value `num` cannot be converted to an `int`. Also if an int-valued method tries to return `null`, a `NullPointerException` will also be thrown.

JUnit

A JUnit test is simply a void method preceded by the @Test annotation. Methods provided by JUnit include assertTrue(...), assertFalse(...), assertEquals(expected,actual), assertNull(...), and assertNotNull(...).

A method preceded by the annotation @Before will be executed before each test in the suite.

Question 4. (10 points) Give a suitable Representation Invariant (RI) and Abstraction Function (AF) for the Menu class describing when the data is valid and the abstract meaning of a valid Menu object.

(a) Representation Invariant

(b) Abstraction Function

Question 5. (18 points) Specification and implementation. Method `getOrderPrice` is supposed to take a list of strings that represent an order, and return the total price of the items in that order. For example, if the menu contains the pairs <"cheeseburger", 349> and <"pepsi", 120>, then `getOrderPrice` should return 818 for the input list {"cheeseburger", "cheeseburger", "pepsi"}. (i.e., 349+349+120). The strings are supposed be ones that are found in the menu.

Give a proper specification and implementation of this method below. Your specification should be complete and the implementation must satisfy the specification. If any parts in the specification are not needed (e.g., requires, modifies, etc.), leave them blank.

```
/**
 * Return the total cost of the items in the list order.
 *
 * @param
 *
 * @requires
 *
 * @throws
 *
 * @modifies
 *
 * @effects
 *
 * @return
 *
 */
public int getOrderPrice(List<String> order) {









}
```

Question 6. (8 points) Testing. The following JUnit test checks if the getOrderPrice method is working properly. However there are several *stylistic* issues with it, even though it does compile and execute, and the test passes. Circle two problems with this test code and ***briefly*** describe what's wrong and what should be changed.

There are more than two issues that could be improved. You only should explain two of them.

```
public class TestSuite {

  private static final Menu TEST_MENU = new Menu();

  @Before
  public void setup() {
    TEST_MENU.addItem("Cheeseburger" , 200);
    TEST_MENU.addItem("Pizza" , 500);
    TEST_MENU.addItem("Water" , 120);
  }

  // All issues that need fixing appear beyond this point:

  @Test
  public void test1 () {
    List<String> order = new ArrayList<String >();
    double total = TEST_MENU.getOrderPrice(order);
    assertTrue(total == 0);


    order.add("Cheeseburger");
    order.add("Pizza");
    total = TEST_MENU.getOrderPrice(order);
    assertTrue(total == 700);

    order.add("Water");
    total = TEST_MENU.getOrderPrice(order);
    assertTrue( total ==820);
  }
}
```

Question 7. (18 points) Specifications and implementations. We would like to add a method `int getPrice(String item)` to our Menu class. This method should return the price of the requested item, but it needs to deal with the cases where the item is not stored in the menu. Here are four possible specifications for this method.

```
/** SPEC A
  * @requires item is not null and item appears in the menu
  * @return the price of the item
  */

/** SPEC B
  * @requires item is not null
  * @return the price of the item if it appears in the menu, or
  *         0 if the item is not found in the menu
  */

/** SPEC C
  * @requires item is not null
  * @return the price of item
  * @throws NoSuchElementException if item is not found in menu
  *       (this is a standard Java library unchecked exception)
  */

/** SPEC D
  * @return the price of the item if it appears in the menu, or
  *         0 if the item is not found in the menu
  * @throws NullPointerException if item is null
  */
```

(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 an 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

Question 7. (cont.) Now, here are four possible implementations of `getPrice`:

```
// Implementation 1:
public int getPrice(String item) {
  return items.get(item);
}
// Implementation 2:
public int getPrice(String item) {
  assert(item != null);
  Integer price = items.get(item);
  if (price == null) {
    throw new NoSuchElementException(); // unchecked exception
  }
  return price;
}
// Implementation 3:
public int getPrice(String item) {
  if (item == null) {
    throw new NullPointerException();   // also unchecked
  }
  return items.get(item);
}
// Implementation 4:
public int getPrice(String item) {
  assert(item != null);
  Integer price = items.get(item);
  if (price == null) {
    return 0;
  } else {
    return price;
  }
}
```

(b) (12 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				