CSE 331 19sp Midterm Exam 5/6/19

Name _____ UW ID # _____

There are 15 questions worth a total of 120 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.

Please be sure to write your answers in the spaces provided.

- **Do not write on the backs of pages – they will not be scanned**
- **Do not write on pages that say "do not write on this page"** (Those pages should be removed from the test and placed in the recycle pile)

An extra blank page is provided at the end if you need additional space for answers.

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

For all of the questions involving proofs, assertions, invariants, and so forth, you should assume that all numeric quantities are unbounded integers (i.e., overflow cannot happen) and that integer division is truncating division as in Java, i.e., 5/3 evaluates to 1.

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.

Score _____ / 120

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

Remember: For all of the questions involving proofs, assertions, invariants, and so forth, you should assume that all numeric quantities are unbounded integers (i.e., overflow cannot happen and there are no fractional parts to numbers) and integer division is truncating division as in Java, i.e., 5/3 => 1.

Question 1. (12 points) (Backward reasoning) A traditional warmup question. 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 = x + 1;

{ ____________________________________ }
z = 2 * y;

{ |z| > 6 }
```

(b) (7 points)

```
{ ____________________________________________ }
if (y > 5) {

   { ____________________________________________ }
   z = 2 * y;

   { ____________________________________________ }
} else {

   { ____________________________________________ }
   z = 2 + y;

   { ____________________________________________ }
}

{ z > 0 && z < 5 }
```

Question 2. (16 points) Fibonacci! Recall that the Fibonacci numbers fib(k) are defined as fib(0) = 0, fib(1) = 1, and fib(k) = fib(k-1) + fib(k-2) for k ≥ 2. The following method is alleged to return fib(n). Write a suitable invariant and appropriate assertions to prove that it works correctly. You should assume that the method works correctly for n < 2 and do not need to handle that case. Provide the correct assertions and proof for n ≥ 2.

```
/** @return fib(n) for n >= 0. @requires n >= 0. */
public int fib(int n) {
   if (n < 2) return n;  // base case - ignore in proof
   { n ≥ 2 }
   int k = 1;
   int fibk = 1;
   int fibprev = 0;

   { inv: _____}

   while(k < n) {

      {_____}

      int fibnext = fibk + fibprev;

      {_____}

      fibprev = fibk;

      {_____}

      fibk = fibnext;

      {_____}

      k = k + 1;

      {_____}

   } // end of loop

   { post: _____}



   return fibk;
}
```

Question 3. (9 points) Here are three specifications and three methods that might implement them. Only the parts of the specifications that are different are shown. All of the specifications should include `@param amount`, but that is omitted to save space.

Spec. A:
```
/** @effects decrease balance by amount */
```

Spec. B:
```
/** @requires amount >= 0 and amount <= balance
 *  @effects decrease balance by amount */
```

Spec. C:
```
/** @throws InsufficientFundsException if balance < amount
 *  @effects decreases balance by amount */
```

Impl. 1:
```
void withdraw(int amount) {
    balance = balance - amount;
}
```

Impl 2:
```
void withdraw(int amount) {
    if (balance >= amount) {
        balance = balance - amount;
    }
}
```

Impl 3:
```
void withdraw(int amount) {
    if (amount < 0) {
        throw new IllegalArgumentException();
    }
    balance = balance - amount;
}
```

In the following grid, place an X in the square if the given implementation satisfies the give specification. If the implementation does not satisfy the specification, leave the square blank.

	Spec A	Spec B	Spec C
Impl 1			
Impl 2			
Impl 3			

Question 4. (14 points, 2 each) `equals` and method calls. One of the summer interns who does not know Java very well has been playing around trying to define classes for things made at a bakery. The intern is completely baffled by the behavior of this code, which attempts to define equality for cakes and chocolate cakes. (Yes, this code does not define `equals` correctly, but the question is asking about what actually happens given the code that's here. We will also leave aside the philosophical question of whether other cakes can ever be equal to chocolate cakes in real life.) The code does compile and execute without crashing.

```
public class Cake {
  protected int size;  // visible in subclasses but not
                       //    to outside clients

  public Cake(int size) {
    this.size = size;
  }

  public boolean equals(Cake other) {
    return this.size == other.size;
  }
}

public class ChocolateCake extends Cake {
  private String kind;   // kind of chocolate

  public ChocolateCake(int size, String kind) {
    super(size);
    this.kind = kind;
  }

  public boolean equals(ChocolateCake other) {
    return this.size == other.size &&
           this.kind.equals(other.kind);
  }

  public static void main(String[] args) {
    Cake cake1 = new Cake(1);
    Cake cake2 = new Cake(1);
    Object objectCake1 = (Object) cake1;
    Object objectCake2 = (Object) cake2;
    ChocolateCake chocolateCake =
                    new ChocolateCake(1, "dark chocolate");
    // answer questions on the next page
    // about code inserted here.
  }
}
```

Remove this page from the exam and use it to answer the next question. Do not write on this page or include it with the rest of the exam when you turn it in.

Question 4. (cont.) For each line of code below, indicate what happens if it is inserted by itself at the end of the main method on the previous page and executed. For each one, indicate which method is called during execution (`Object.equals`, `Cake.equals`, or `ChocolateCake.equals`) and whether the method call returns true or false. Circle the correct answers.

(a) `cake1.equals(cake2);`

Class whose `equals` method is executed: `Object` `Cake` `ChocolateCake`

Result: `true` `false`

(b) `cake1.equals(chocolateCake);`

Class whose `equals` method is executed: `Object` `Cake` `ChocolateCake`

Result: `true` `false`

(c) `chocolateCake.equals(cake1);`

Class whose `equals` method is executed: `Object` `Cake` `ChocolateCake`

Result: `true` `false`

(d) `objectCake1.equals(cake1);`

Class whose `equals` method is executed: `Object` `Cake` `ChocolateCake`

Result: `true` `false`

(e) `objectCake1.equals(chocolateCake);`

Class whose `equals` method is executed: `Object` `Cake` `ChocolateCake`

Result: `true` `false`

(f) `objectCake1.equals(cake2);`

Class whose `equals` method is executed: `Object` `Cake` `ChocolateCake`

Result: `true` `false`

(g) `cake1.equals(objectCake1);`

Class whose `equals` method is executed: `Object` `Cake` `ChocolateCake`

Result: `true` `false`

Consider the following class that represents items that are stored in a warehouse. A few of the methods in this class are provided below. Answer questions about this class on the following pages. (This code does compile without errors.)

```
/** A StockItem is a mutable object that represents an item
 *  in a warehouse inventory.  The information in a StockItem
 *  includes the name of the item, a category for the item,
 *  the location in the warehouse where it is stored, and the
 *  number of copies of this item currently in the warehouse.*/
public class StockItem {
  // instance variables
  private String name;
  private int quantity;
  private String category;
  private String location;

  // creators
  /** construct a new StockItem with given properties */
  public StockItem(String name, int quantity, String category,
                                                 String location) {
    this.name = name;
    this.quantity = quantity;
    this.category = category;
    this.location = location;
  }

  // observers
  public String getName() { return name; }
  public int getQuantity() { return quantity; }
  public String getCategory() { return category; }
  public String getLocation() { return location; }

  // mutator
  public void setQuantity(int q) { quantity = q; }

  // equals
  /** return true if this StockItem is equal to o */
  @Override
  public boolean equals(Object o) {
    if ( !(o instanceof StockItem) )
      return false;
    StockItem other = (StockItem)o;
    return this.name.equals(other.name) &&
           this.category.equals(other.category) &&
           this.location.equals(other.location);
  }
```

Remove this page from the exam and use it to answer the following questions. Do not write on this page or include it with the rest of the exam when you turn it in.

Question 5. (8 points, 2 each) (hashCode) Since our `StockItem` class includes an `equals` method, we need to provide a suitable `hashCode` method to go with it. Here are four possible `hashCode` implementations. Each of them compiles. For each one you should indicate whether the implementation satisfies the contract (specification) for `hashCode` given the existing `equals` method in `StockItem` and, if it does, whether it is a good, adequate, or poor choice for `hashCode`. Put an X next to the best answer.

(a)
```
public int hashCode() {
        return this.name.hashCode();
   }
```
_____ Incorrect (does not satisfy the contract for `hashCode`)

_____ Correct but poor quality

_____ Correct with adequate quality (not terrible but not particularly great)

_____ Correct with good/high quality

(b)
```
public int hashCode() {
        return 1;
   }
```
_____ Incorrect (does not satisfy the contract for `hashCode`)

_____ Correct but poor quality

_____ Correct with adequate quality (not terrible but not particularly great)

_____ Correct with good/high quality

(c)
```
public int hashCode() {
        return this.name.hashCode() ^ this.quantity;
   }
```
_____ Incorrect (does not satisfy the contract for `hashCode`)

_____ Correct but poor quality

_____ Correct with adequate quality (not terrible but not particularly great)

_____ Correct with good/high quality

(d)
```
public int hashCode() {
        return this.name.hashCode() ^ this.category.hashCode() ^
          this.location.hashCode();
   }
```
_____ Incorrect (does not satisfy the contract for `hashCode`)

_____ Correct but poor quality

_____ Correct with adequate quality (not terrible but not particularly great)

_____ Correct with good/high quality

The next questions use the `StockItem` class from the previous question. Suppose we now define a class to hold a collection of `StockItem`s. Here is the start of the class definition:

```
/** A collection of StockItems {s1, s2, ..., sn}.  No two
 *  StockItems have the same name. */
public class Stock {
  // instance variable
  Private List<StockItem> items;  // StockItems in this
                                  //    Stock collection

  ...
}
```

Question 6. (6 points) (JavaDoc and specs) Our `Stock` class contains the following constructor which, alas, is missing the usual CSE 331-style specification. Complete the JavaDoc comment so it properly specifies the operation of this constructor with appropriate JavaDoc tags and fields. (For CSE 331-specific tags like requires, you can use either `@requires` or `@spec.requires` – both will receive full credit. Also, you almost certainly won't need all this space.)

```
  /** construct new empty Stock collection
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   *
   */
  public Stock() {
    items = new ArrayList<StockItem>();
  }
```

Question 7. (11 points) RI, AF, and checkRep (Hint: these are pretty simple – don't panic if the answers are short.) More questions about the `Stock` class, from the previous page. Your answers should be consistent with the instance variable and constructor code given there.

(a) (4 points) Give a suitable representation invariant (RI) for class `Stock`.

(b) (3 points) Give a suitable abstraction function (AF) for class `Stock`.

(c) (4 points) Complete the implementation of `checkRep()` for class `Stock`.

```
// terminate execution with an assertion failure if a violation
// of the rep invariant is discovered, otherwise return silently
private void checkRep() {

}
```

Question 8. (8 points) (Another specification) Our `Stock` class has the following method, which returns a list of all of the `StockItems` whose location matches the method parameter. As before, complete the JavaDoc comment so it properly specifies the operation of this method using CSE 331 specification conventions.

```
/** Return a list of StockItems whose location match the
 *  the method parameter.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */
public List<StockItem> getItemsAtLocation(String location) {
  List<StockItem> result = new ArrayList<StockItem>();
  for (StockItem item: items) {
    if (item.getLocation().equals(location)) {
      result.add(item);
    }
  }
  return result;
}
```

Question 9. (10 points) Representation exposure. Take another look at the `getItemsAtLocation` method on the previous page.

(a) (2 points) Does this method create any potential representation exposure problems, either for this class or any other class? (circle)

Yes No

(b) (5 points) Give a brief, but complete explanation and justification for your answer to part (a). A few sentences should be sufficient. Answers that are correct but excessively long will not necessarily receive full credit.

(c) (3 points) If there are any potential representation exposure problems, give a brief but complete description of how to fix them (you do not need to write actual Java code). If there are no potential representation exposure problems, just write "none" to receive full credit for this part of the question.

Question 10. (12 points, 3 each) Testing. Describe four distinct black-box tests that could be used to verify that the `getItemsAtLocation` method from the previous problem works properly. Each test description should describe the test input and expected output. For full credit each test should be different in some significant way from the other tests (think about boundary conditions and subdomains, etc.). You **should not** provide JUnit or other code, just a clear, precise description of each test, and your descriptions should be a few lines each, at most. If you want to write a specific StockItem as part of a test you can use something like (name, quantity, category, location), i.e., (gum, 17, food, bin42), but you don't have to do this.

(a) Input or test setup:

Expected output:

(b) Input or test setup:

Expected output:

(continued on next page)

Question 10. (cont.)

(c) Input or test setup:

Expected output:

(d) Input or test setup:

Expected output:

Some short-answer questions to wrap up.

Question 11. (3 points) Test metrics. There are several metrics that are used to measure test coverage. In alphabetical order, some of the coverage metrics we looked at were *branch*, *loop*, *path*, and *statement* coverage. One tradeoff between these is that the most comprehensive metrics are also the most expensive. Write a list of these four metrics **from *least* to *most*** expensive and comprehensive.

Question 12. (3 points) Tests for bugs. One of the guidelines for testing is that when a bug is discovered, you should create a test for it and add it to the test suite permanently. The question is why should we retain this test forever? After all, once we've fixed the bug we no longer need the test, do we? (Be brief!)

Queston 13. (3 points) Preconditions in specs. Suppose are writing a method and we have a choice between using `@throws IllegalArgumentException if x<0` and `@requires x>=0` in the method specification. Neither option is significantly harder or more expensive to implement. Which is the better choice to include if the method is going to be included in a widely distributed library, and why? (briefly!)

Question 14. (3 points) Checked vs. unchecked exceptions. Java makes a distinction between checked exceptions (things like `FileNotFoundException`) and unchecked ones (like `NullPointerException`). Checked exceptions are required to be included in a `throws` clause in a method heading if the method can throw them, while unchecked exceptions do not have this requirement. Why did the Java designers decide to treat checked exceptions this way rather than treating them like unchecked exceptions, which do not need to be declared as part of the method heading? (briefly!!)

Question 15. (2 free points) (All reasonable answers receive the points. All answers are reasonable as long as there is an answer. ☺)

(a) (1 point) What question were you expecting to appear on this exam that wasn't included?

(b) (1 point) Should we include that question on the final exam? (circle or fill in)

Yes

No

Heck No!!

$!@$^*% No !!!!!

No opinion / don't care

None of the above. My answer is _____.

Additional space for answers if needed. Please indicate clearly which questions you are answering here, and also be sure to indicate on the original page that the rest of the answer can be found here.