CSE 331 22wi Midterm Exam 2/8/22 Sample Solution

Remember: For 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, 6 each) Forward reasoning with (maybe) bugs! Here are two code sequences with assertions inserted using forward reasoning. But there may be problems: the sequences may contain an error in the reasoning – or maybe not. In each code sequence, do one of the following:

- If there are no errors in the assertions, write "correct" to the side of the problem.
- If there are one or more errors in the assertions, circle the *first* assertion that is incorrect. An assertion is incorrect if either it is a logic error or if it is weaker than the strongest assertion that could appear at that position in the code.
- You do not need to justify your answers, just circle the first buggy assertion. (Note that there may be further errors later in the sequence because of previous errors. You only should circle the first error in the sequence if there is one.)

You should assume that the actual code is correct – the question is about the logical reasoning. Also, recall that $\wedge$ is the symbol for logical "and".

(a)
```
{x = z}
x = 2*x;
{x = 2z}
z = 2*z;
[✓ {x = 4z}]      (should be { x=z })
x = 2*x – 3;
{x = 8z - 3}
```

(b)
```
{x = 2y ∧ z > 0}
x = x + 1
{x = 2y + 1 ∧ z > 0}
x = x - z;
{x = 2y - z + 1 ∧ z > 0}
z = z - 1;
{ x = 2y - z ∧ z > -1}
y = y - z;
[✓ {x = 2y ∧ z > -1}]      (should be { x = 2y+z ∧ z > -1 } )
```

Notes: all that was required was to circle the correct answer. The "should be" comments above are for information only. Also, we tried to provide partial credit when possible if an earlier or later assertion was selected because of simple algebra mistakes or similar errors.

Question 2. (14 points) (Backward reasoning). We're trying to debug some code using backwards reasoning. The code fragment we're working on has an assertion that indicates that when execution reaches the beginning of this sequence, we know that the assertion {y>5} holds. After execution, the postcondition {x+y>5} should hold. But something could be wrong with the code. We want to use backwards reasoning to see if there is a bug.

(a) (12 points) Find the weakest precondition for the sequence of statements below by starting with the postcondition and reasoning backwards to the beginning. Your weakest precondition should appear in the first blank space when you're done. Write appropriate assertions in each line and simplify your final answer if possible. (Note that { y>5 } is the known assertion that appears at the beginning of the code we've found, but it might, or might not, be the weakest precondition for the if statement – that is part of what we're investigating.)

```
{ y > 5 }
{  (y%2=0 ∧ y%2+y > 5) ∨ (y%2!=0 ∧ 2y > 7) } =>
     { (y  >  5) ∨ (y%2=1 ∧ 2y>7) } => {(y>5) ∨ (y>=5)} => {y>=5}
if (y % 2 == 0) {
  { y%2 + y > 5 }
  x = y % 2;
  { x + y > 5 }
} else {
  { y-2 + y > 5 } => { 2y > 7 }
  x = y - 2;
  { x + y > 5 }
}
{ x + y > 5 }
```

(b) (2 points) Now that we've figured out the weakest precondition for the if statement that is needed to guarantee the post-condition, is the known { y > 5 } assertion at the beginning of the code sufficient to guarantee that the weakest pre-condition holds there? (Or another way to ask the question is: is { y>5 } *if-statement* { x+y >5 } a valid Hoare triple?) You only need to answer yes or no.

Yes. (The {y>5} assertion at the beginning is stronger than (implies) the weakest precondition {y>=5}.)

Question 3. (14 points) A loopy invariant. One of our colleagues is trying to fix a possibly buggy search method. They have heard that you know all about these fancy "loop invariants" from CSE 331 and are hoping you might be able to help them.

Here is the code and the loop invariant they have come up with so far:

```
public int findMinLoc(int[] arr) {
  int i = 1;
  int minLoc = 0;
  // inv: arr[minLoc] is min value in arr[0..i-1]
  while (i != arr.length) {
    { inv }
    if (arr[i] < arr[minLoc]) {
      {arr[minLoc] = min arr[0..i-1} ∧ arr[i]<arr[minloc]}
      minLoc = i;
      {arr[minLoc] = min arr[0..i]}
    } else {
     {arr[minloc] = min arr[0..i-1]∧arr[i]>=arr[minloc]=>
      {arr[minLoc] = min arr[0..i]}
      i = i + 1;
      {arr[minLoc] = min arr[0..i-1]}
    }
    {(arr[minLoc] = min arr[0..i])
      ∨ arr[minLoc] = min arr[0..i-1]}
  }
  {i == arr.length ∧ (arr[minLoc] = min in arr[0..i])
    ∨ arr[minLoc] = min in arr[0..i-1]} =>
  {(arr[minLoc] = min in arr[0..arr.length])
    ∨ arr[minLoc] = min in arr[0..arr.length-1]}
  // post:
  // arr[minLoc] is min value in arr[0..arr.length-1]
  return minLoc;
}
```

Your job is to discover whether this code can be proved correct using the given invariant and the loop proof techniques (invariants, assertions, postconditions, etc.) we have covered in class. Justify your answer by annotating the code above with assertions in enough places to support your answer (i.e., you don't have to include every possible assertion, but include enough to clearly support your conclusion) and then explain your conclusion briefly.

This question turned out to be a bit harder to grade than expected. Answers were evaluated to see how well they demonstrated understanding of the various proof rules for loops, conditional statements, and assignments. The question deliberately did not require including all possible assertions to save time during the exam, so sometimes we had to use our best judgement to decide when omitted steps were straightforward enough that they didn't need to be included compared to places where too much was omitted and the answer didn't show clear understanding of the proof issues involved.

Once all the assertions are added to the code, we see that the assertion after the `if` statement is not the same as the loop invariant. Our normal proof techniques do not work cleanly in this case.

It turns out that the program "works", in that it ultimately gets the right answer, but almost by accident. When the `if` condition is true, variable `minLoc` is updated, but `i` is not increased, which is why we do not get the loop invariant as the `if`-statement postcondition after executing that branch of the `if` statement. The next time around the loop, `minLoc` will not be updated, but `i` will be increased, so eventually the search will reach the end of the array.

To actually prove that the method returns the right result, we would have to carefully show that the loop does make progress, at least on every other iteration, and that it eventually terminates with the right result. We did not focus on termination proofs in CSE331 this quarter, so that proof would be beyond what our basic correctness logic can show.

A rite of passage when people get their first apartment is a trip to Ikea to get some furniture. Of course, Ikea furniture comes in kits to be assembled. The next several questions concern classes `Part` and `PartList` which can be used to store information about the collection of parts in a kit for a piece of furniture. The code for these classes is included on separate pages at the end of the exam. You should **remove those pages** from the exam and use them while answering these questions.

Question 4. (12 points) Let's look at class `Part`. First, we need to complete the `equals` and `hashCode` methods. Here are several possible `return` statements that could appear in `equals` in place of the TODO comment.

```
E1:   return this.name.equals(p.name);
E2:   return this.weight == p.weight&&
                   this.quantity == p.quantity;
E3:   return this.name.equals(p.name)&&
                   this.weight == p.weight;
E4:   return true;
```

Now here are several possibilities for completing method `hashCode` replacing the TODO there (as with the code above, all of these choices compile with no errors):

```
H1:   return 331;
H2:   return this.name.hashCode();
H3:   return this.name.hashCode() + 31*this.quantity;
H4:   return this.name.hashCode() + 31*(int)this.weight;
```

(a) (10 points) In the following table, put an X in the space if the given hash function from the above list is *consistent with* (i.e., satisfies the requirements to be used with) the given equality relation from the first list. Your answer should ignore whether or not the `equals` relation actually is a correct equals method that satisfies the required properties for equality. Just mark an X where the `hashCode` is consistent with (i.e., satisfies the required properties for `hashCode`) given that particular definition of `equals`.

	E1	E2	E3	E4
H1	X	X	X	X
H2	X		X	
H3				
H4			X	

(b) (2 points) Suppose we implement `equals` using the statement `return this.name.equals(p.name)&& this.weight == p.weight;` (E3 above). Given the above possibilities for `hashCode` (H1-H4) and this choice for `equals`, what choice for `hashCode` would be best for the `Part` class? Pick one of H1-H4 and write your answer below. You do not need to justify your answer.

H4

Question 5. (12 points) ADT/RI/AF. Now on to the `PartList` class. The first thing we need to do is be sure we understand the ADT that this class represents. To do that we need to provide an abstract description of the class, a rep invariant, and an abstraction function. Your answers should be consistent with the informal description of the class and the given instance variables and code included with the supplied `PartList` class.

(a) (3 points) Give a suitable description of the `PartList` ADT and its abstract value(s), as would normally appear in the JavaDoc comment right above the class definition. (Hint: the answer might be quite short.)

A PartList is a mutable, unordered collection of non-null Part objects representing the parts in a furniture kit. No two Parts in the PartList may have the same name. A PartList value can be denoted as a set { p1, p2, …, pn }, where each pi is a Part.

(b) (5 points) Give a suitable representation invariant (RI) for `PartList`.

parts != null &&
no element in parts is null &&
for 0 <= i, j, < parts.size(), if i != j then
parts.get(i).getName.equals(parts.get(j).getname()) is false.

(Note that the "no elements are null" restriction is needed since code in PartList includes method calls on elements of the list, and that code will fail if an element of the list is null. For the "no duplicates" part of the rep invariant, answers that were less formal and said something like "if i!=j then parts i and j do not have the same name" were perfectly fine.)

(c) (4 points) Give a suitable abstraction function (AF) for `PartList`.

AF(this) = the set { p1, p2, …, pi } where each pi is an element of the list parts.

Question 6. (12 points) As usual, the instructor who wrote this question still hasn't learned after all these years how to provide proper specifications for things. Below, fill in a correct CSE 331-style specification for the `addPart` and the `getPart` methods of class `PartList`. Your answer should be consistent with the given code and what it does when executed. For CSE 331-specific custom tags, you can write `@spec.xyz` or just `@xyz` – whichever you prefer.

```
/** Add a new part to this PartList provided that no part
 *  already in the list has the same name.
 *
 * @param p the new Part to be added to this
 *
 * @requires p!=null, and p.getName() is different from names
 *           of other parts already present in this.
 *
 * @modifies this
 *
 * @effects p is added to this
 *
 *
 *
 *
 */
public void addPart(Part p) { ... }
```

Note that the precondition (@requires) is needed for the existing code in addPart work without failures.

```
/** Return information about the part with the given name in
 * this PartList, or return null if not found.
 *
 * @param name the name of the desired part in this PartList
 *
 * @return the part p in this where p.getName().equals(name),
 *         or null if no part in this has the requested name.
 *
 *
 *
 *
 *
 *
 *
 *
 */
  public Part getPart(String name)
```

Question 7. (10 points, 2 each) Representation exposure. As often happens, the client for our `PartList` class has requested a change, this time to add a method that returns a list of the `Parts` in the `PartList`. Here are four proposed implementations of a new `getParts` method. For each proposed implementation, circle yes if there is a representation exposure problem that could allow the client to do something that would invalidate the rep invariant of the `PartList` class. Circle no if the method does not expose the rep in a way that could cause this problem.

(a)
```
public List<Part> getParts() {
    return parts;
}
```

Can cause bugs due to rep exposure: [✓ yes] no

(b)
```
public List<Part> getParts() {
    return new ArrayList<Part>(parts);
}
```

Can cause bugs due to rep exposure: yes [✓ no]

(c)
```
public List<Part>GetParts() {
    List<Part> copy = new ArrayList<Part>();
    for (Part p: parts) {
      copy.add(new Part(p.getName(), p.getWeight(),
                        p.getQuantity()));
    }
    return copy;
}
```

Can cause bugs due to rep exposure: yes [✓ no]

(d)
```
public List<Part> getParts() {
    return Collections.unmodifiableList(parts);
```

Can cause bugs due to rep exposure: yes [✓ no]

(e) Of the four methods above, which would be the <u>best</u> choice (correct and efficient and with no potential rep exposure bugs) if we want method `getParts` to return a list containing the `Parts` in the `PartList` at the time the method was called, and which will not change over time unless the client chooses to modify the returned list? Below, write a single letter `a` through `d` giving your choice. You do not need to justify your answer.

`b` (`c` also returns a correct result that will not change after it is returned to the client, but it is more expensive because it makes unnecessary copies of the immutable `Part` objects.)

Question 8. (10 points, 2 each) Overloading, overriding, and `equals`. This question is about the following `main` method that uses the `Thing` and `Holder` classes printed on the last separate code page at the end of the exam. Detach that page and use it to answer this question. All of the code compiles and runs without errors.

```
  public static void main(String[] args) {
    Thing thingA = new Thing(8);
    Thing thingB = new Thing(8);
    Holder holdA = new Holder(8,-29);
    Holder holdB = new Holder(7,70);
    Object objThingA = thingA;
    Object objThingB = thingB;
    Object objHoldA = holdA;
    Object objHoldB = holdB;
    ______________________ ;  // insert code from below here
  }
}
```

For each line of code below, indicate what happens if it is inserted by itself at the end of the `main` method above and then the program is executed. For each one, circle the correct answers to indicate which method is called during execution (`Object.equals`, `Thing.equals`, or `Holder.equals`) and whether the method call returns `true` or `false`. Circle only the class of the first `equals` method called, even if that method calls another one.

(a) `System.out.println(thingB.equals(holdA));`

`equals` method executed: `Object` [✓ `Thing`] `Holder` Result: [✓ `true`] `false`

(b) `System.out.println(objThingA.equals(thingA));`

`equals` method executed: [✓ `Object`] `Thing` `Holder` Result: [✓ `true`] `false`

(c) `System.out.println(holdB.equals(thingB));`

`equals` method executed: `Object` `Thing` [✓ `Holder`] Result: `true` [✓ `false`]

(d) `System.out.println(thingA.equals(objThingA));`

`equals` method executed: [✓ `Object`] `Thing` `Holder` Result: [✓ `true`] `false`

(e) `System.out.println(thingB.equals((Thing) objHoldB));`

`equals` method executed: `Object` [✓ `Thing`] `Holder` Result: `true` [✓ `false`]

Answers that identified the wrong class but then circled a true/false answer that was consistent with executing the `equals` method in the identified class received partial credit.

Question 9. (4 free points) (All reasonable answers receive the points. All answers are reasonable as long as there is an answer. ☺)

(a) (2 point) What question were you expecting to appear on this exam that wasn't included?

Implement a Java compiler. Be sure to include all necessary library classes and methods that are found the standard Java 11 distribution.

(b) (2 points) Should we include that question on the final exam? (circle or fill in)

Yes

No

Heck No!!

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

[✓ Yes, yes, it *must* be included!!!]

No opinion / don't care

None of the above. My answer is ________________________________.

CSE 331 22wi Midterm Exam 2/8/22 Sample Solution

Code for classes `Part` and `PartList`. Remove these pages from the exam and return them for recycling when you are done. This code is used in several questions in the exam. Some parts of the code are incomplete or missing, and the questions address those issues. Except for the missing pieces, all of the code here does compile and work as intended.

Class **`Part`**: an immutable object representing a part in a furniture kit. An example of an abstract `Part` would be ("nail", 1.2, 4), meaning there are 4 nails weighing 1.2 grams each in this `Part`. A part must have a positive (i.e., >0) quantity and weight.

```
public class Part {
  // instance variables
  private String name;        // name of this part
  private double weight;      // weight in grams of a single
                              // individual part
  private int quantity;       // total number of this part in the
                              // furniture kit

  // Creators
  /** construct a new Part with given properties */
  public Part(String name, double weight, int quantity) {
    this.name = name;
    this.weight = weight;
    this.quantity = quantity;
  }

  // observers
  public String getName() { return name; }
  public double getWeight() { return weight; }
  public int getQuantity() { return quantity; }

  // equals/hashCode
  /** return true if this Part is equal to o */
  @Override
  public boolean equals(Object o) {
    if ( !(o instanceof Part) )
      return false;
    Part p = (Part)o;
    return /* TODO: figure out what to put here */;
  }

  @Override
  public int hashCode() {
    return /* TODO: figure out what to put here */;
  }

}  // end class Part
```

Class **`PartList`**: a mutable unordered collection of `Part` objects representing the parts in a furniture kit. No two parts in the `PartList` may have the same name.

```
public class PartList {
  // instance variables
  private List<Part> parts;

  /** construct new empty PartList */
  public PartList() {
    parts = new ArrayList<Part>();
  }

  /** Add a new part to this partlist provided that no part
   *  already in the list has the same name.    */
  public void addPart(Part p) {
    parts.add(p);
  }

  /** return information about the part with the given name in
    * this PartList, or return null if not found */
  public Part getPart(String name) {
    for (Part p: parts) {
      if (p.getName().equals(name)) {
        return p;
      }
    }
    return null;
  }

  /** return the total weight of all Parts in this PartList */
  public double getWeight() {
    double weight = 0.0;
    for (Part p: parts) {
      weight += p.getWeight() * p.getQuantity();
    }
    return weight;
  }

}  // end of PartList
```

Code for `Thing`/`Holder` classes used in `equals` overloading/overriding question. Remove this page from the exam and return it for recycling when you are done.

Notice that the parameters to the `equals` methods have unusual types (possibly not what they should be, but the question is about what happens given this code as written).

```
/** A Thing object holds an integer value. */
class Thing {
  private int t;
  public Thing(int t) {
    this.t = t;
  }
  public boolean equals(Thing o) {
    if (!(o instanceof Thing)) {
      return false;
    }
    Thing thing = (Thing) o;
    return this.t == thing.t;
  }
}

/** A Holder is a Thing plus a second integer value. */
class Holder extends Thing {
  private int h;
  public Holder(int t, int h) {
    super(t);
    this.h = h;
  }
  public boolean equals(Thing o) {
    if (!(o instanceof Thing)) {
      return false;
    }
    if (!(o instanceof Holder)) {
      return super.equals(o);
    }
    Holder holder = (Holder) o;
    return super.equals(holder) && this.h == holder.h;
  }
}
```