CSE 331 23wi Midterm Exam 2/7/23

Name ____________________________________ UW netid _______________

There are 9 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. However, you may have a single 5x8 notecard with any hand-written notes you wish on both sides.

The longer code listings are printed on separate pages at the end of the exam. You should **remove those last two sheets of paper from the exam** and use them while answering questions. Leave those pages on the recycle pile when you turn in your exam at the end of the hour.

Please do NOT remove any pages from the middle of the exam.

There is a blank sheet of paper at the end with extra space for your answers if you need more room. It is after all the questions but before the detachable pages with the code.

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 _________________ / 100

1. ______ / 12	6. ______ / 9
2. ______ / 14	7. ______ / 12
3. ______ / 14	8. ______ / 15
4. ______ / 10	9. ______ / 2
5. ______ / 12	

Reminder: 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, 5/-3 => -1.

Question 1. (12 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 < 3} holds. After execution, the postcondition {x+y<1} should hold. We want to use backwards reasoning to see if the code is correct.

(a) (10 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 (the weakest precondition) if possible, but be sure to show your work for any simplifications in case errors creep in. (Note that { y<3 } is a known assertion that is true at the beginning of the code we've found, but it might, or might not, be the weakest precondition for the if statement.)

```
{ y < 3 }


{ ________________________________________________ }
if (y % 2 == 0) {
   { ________________________________________________ }
   x = y - 1;
   { ________________________________________________ }
} else {
   { ________________________________________________ }
   x = y + 2;
   { ________________________________________________ }
}
{ x + y < 1 }
```

(b) (2 points) Now that we've figured out the weakest precondition for the if statement needed to guarantee the post-condition, is the known { y < 3 } 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 < 3 } *if-statement* { x+y < 1 } a valid Hoare triple?) You only need to answer yes or no.

Question 2. (14 points) A loopy question. One of our colleagues really doesn't like recursion so they have written a method to calculate *n*! = 1 * 2 * 3 * … * *n* using iteration. The code *seems* to work, but we want to be really sure it is correct. Your job is to prove that the code really does compute *n*! correctly and return it as the result. You will need to design a suitable loop invariant and write assertions between the statements as needed to prove that it calculates and returns the correct result. If needed, you can add a suitable precondition for the method, but that is not required if you don't need it. The loop postcondition and places to write assertions are provided for you to use as needed.

```
/** Returns the value of n! = 1*2*3*...*n */

public static int fact(int n) {

  int ans = 1;

  int k = 2;

  { ______________________________________________ }

  { inv: ______________________________________ }

  while (k != n+1) {

    { ______________________________________________ }

    ans = ans * k;

    { ______________________________________________ }

    k = k + 1;

    { ______________________________________________ }

  }

  { ______________________________________________ }

  { post: ans = n! }

  return ans;

}
```

The next several questions concern classes `Task` and `TaskAssignments`. 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 3. (14 points) Let's look at class `Task`. 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.description.equals(t.description)&&
          this.time == t.time;
E2:   return this.time == t.time;
E3:   return this.description.length() ==
                                t.description.length();
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 (int)(Math.random() * time);
H2:   return description.length();
H3:   return description.hashCode();
H4:   return 31 * time + description.hashCode();
H5:   return 331;
```

(a) (12 points) In the following table, put an X in the space if the given hash function from the above list is *consistent with* the given `equals` method from the first list. Your answer should ignore whether or not the `equals` method actually is a correct equality relation that satisfies the required properties for equality. Just put 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				
H2				
H3				
H4				
H5				

(b) (2 points) Suppose we implement `equals` using the statement `return this.description.equals(t.description)&& this.time == t.time;` (E1 above). Given the above possibilities for `hashCode` (H1-H5) and this choice for `equals`, what choice for `hashCode` would be best for the `Task` class? Write one of H1 to H5 below. You do not need to justify your answer.

Now on to the `TaskAssignments` class. We're looking to use this code to keep track of the many incoming jobs that are needed to manage CSE331 and assign them to TAs and instructors. This software looks promising. Unfortunately, it is somewhat incomplete and only partially implemented, and the implementation we have seems to have some bugs or at least specification / documentation errors. We'll try to clean these up over the next few questions.

Question 4. (10 points) RI/AF. The first thing we need to do is be sure we understand the ADT that this class represents. To do that we need to 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 variable(s) and code included in the existing `TaskAssignments` class. Fortunately, the abstract description has been provided in the starter code and looks to be ok as-is.

(a) (6 points) Give a suitable representation invariant (RI) for `TaskAssignments`.

(b) (4 points) Give a suitable abstraction function (AF) for `TaskAssignments`.

Question 5. (12 points) Specifications. `assignTask` is a method in class `TaskAssignments` that assigns a new task to an employee who is not already busy with another task. Originally, we were given only the code. We do know that it works correctly, so the implementation is not an issue, but it did not have a proper specification. One of the CSE 331 staff members (the instructor) has attempted to provide a proper CSE 331-style JavaDoc specification for this method, but there seem to be some problems.

For this question, your job is to fix the specification of the following method so that it matches the code and specifies it correctly. You should assume this is a method in `TaskAssignments` (the rest of the code for that class is included on a separate page at the end of the exam). Mark your changes and corrections and write in any necessary additions directly on the code below. You do not need to explain your changes, although you can do so if you wish.

```
  /**
   * Assigns the task with the given description and time
   * to the given employee.
   *
   * @param employee name of person to do the given task
   * @param newTask Task to assign to the given employee
   * @requires duration of newTask is positive
   * @spec.modifies this.taskMap
   * @spec.effects adds a Task for this employee if and
   *   only if they are not already assigned a task.
   * @throws IllegalArgumentException if duration (time) of
   *         the task is not positive
   */
 public void assignTask(String employee, Task newTask) {
   if (newTask.getTime() <= 0) {
     throw new IllegalArgumentException(
                          "Task time must be positive");
   }
   if (!taskMap.containsKey(employee)) {
     taskMap.put(employee, newTask);
   }
 }
```

Question 6. (9 points, 3 each) Comparing specifications. It's midterm time so it's almost time to think about registering for courses for spring quarter(!). We've written a method that attempts to register students for classes. The method heading is

```
public Boolean updateSchedule(String netid,
                              List<String> classes);
```

Here are several possible specifications for this method.

Spec A:
@requires netid is the id of a student currently enrolled at UW
@return true if the student with the given netid was successfully enrolled in all of the listed classes and false otherwise

Spec B:
@return true if the student with the given netid was successfully enrolled in all of the listed classes and false otherwise
@throws IllegalArgumentException if netid does not belong to a student currently enrolled at UW

Spec C:
@return true if the student with the given netid was successfully enrolled in all of the listed classes and false if the netid does not belong to a student currently enrolled at UW or if the student with that netid was not able to successfully enroll in all of the listed classes.

Spec D:
@requires netid is the id of a student currently enrolled at UW and classes contains only classes being offered next quarter
@return true if the student with the given netid was successfully enrolled in all of the listed classes and false otherwise

For each of the pairs of specifications listed below, if one of the specifications is stronger than the other, circle the letter of the stronger specification. If neither specification is stronger than the other one, circle "neither"

(a) A B neither

(b) B C neither

(c) D A neither

Question 7. (12 points, 3 each) Testing. In hw2 we proved that the following method correctly computes the value x^y.

```
// return the value x^y.
// pre: x>=0 && y >= 0
public int expt(int x, int y) {
  int z = 1;
  while (y > 0) {
    if (y % 2 == 0) {
      y = y/2; x = x*x;
    } else {
      z = z*x; y = y-1;
    }
  }
  return z;
}
```

However, we've also learned that we should test our code to be sure it works as expected.

For this question, describe four black-box or clear-box tests for this function. Your collection of tests should be such that each line of code in the method is executed at least once by at least one of the tests, and each test should ideally test a separate subdomain of the possible input values that is different from the other tests in some way. For each test, describe the inputs (values of x and y), expected output, and, in one sentence, why you chose to include that particular test in the collection.

(i) input: x = ______ , y = _______ ; expected output = __________
Why included? (one sentence):

(ii) input: x = ______ , y = _______ ; expected output = __________
Why included? (one sentence):

(iii) input: x = ______ , y = _______ ; expected output = __________
Why included? (one sentence):

(iv) input: x = ______ , y = _______ ; expected output = __________
Why included? (one sentence):

Question 8. (15 points, 3 each) The suspiciously familiar but not quite the same overloading, overriding, and `equals` question. This question is about the following `main` method that uses the `Int` and `IntInt` 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) {
    Int inta = new Int(1);
    Int intb = new Int(1);
    IntInt intinta = new IntInt(3,7);
    IntInt intintb = new IntInt(2,4);
    Object oia = inta;
    Object oib = intb;
    Object oiia = intinta;
    Object oiib = intintb;
    ______________________ ;  // 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`, `Int.equals`, or `IntInt.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(oia.equals(intb));`

`equals` method executed: `Object` `Int` `IntInt` Result: `true` `false`

(b) `System.out.println(inta.equals(intintb));`

`equals` method executed: `Object` `Int` `IntInt` Result: `true` `false`

(c) `System.out.println(oia.equals(oib));`

`equals` method executed: `Object` `Int` `IntInt` Result: `true` `false`

(d) `System.out.println(intinta.equals(intintb));`

`equals` method executed: `Object` `Int` `IntInt` Result: `true` `false`

(e) `System.out.println(oiia.equals(intinta));`

`equals` method executed: `Object` `Int` `IntInt` Result: `true` `false`

Question 9. (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 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 ________________________________.

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.

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.

Code for classes `Task` and `TaskAssignments`. 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 **`Task`**: an immutable object that represents a task identified by its description and the expected time needed to perform the task, in minutes. The description of the Task can be any valid **`String`** and time can be any **`int`** value > 0. A typical Task might be: ("midterm exam", 50).

```
public class Task {
  // instance variables
  private final String description; // description of the task
  private final int time;           // expected min. to do task

  /** construct a new Task
    * ... remainder of spec omitted ...
    */
  public Task(String description, int time) {
    assert (time > 0);
    this.description = description;
    this.time = time;
  }

  // observers
  public String getDescription() { return description; }
  public int getTime() { return time; }

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

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

}  // end class Task
```

Class **`TaskAssignments`**: a mutable unordered collection of `Task` objects.

```
/**
 * A TaskAssignments object is a mutable, unordered collection of
 * information about tasks and individuals assigned to complete
 * them.  An element of TaskAssignments is a <worker, task> pair
 * indicating that the given task is being handled by the given
 * worker.  A worker may be assigned only one task at a time,
 * so no two elements in a TaskAssignments object have the same
 * worker name.
 */
public class TaskAssignments {
  // representation
  private Map<String, Task> taskMap;
                           // Collection of <worker, task> pairs
  /**
   * Create a new, empty TaskAssignments object.
   * @spec.effects Creates an empty TaskAssignments
   */
  public TaskAssignments() {
    taskMap = new HashMap<String, Task>();
  }

  // Additional methods for this class are included in the exam
  // questions dealing with this class.

}  // end class TaskAssignments
```

Code for `Int`/`IntInt` 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 Int object holds an integer value. */
class Int {
  private int n;
  public Int(int n) {
    this.n = n;
  }
  public boolean equals(Int o) {
    if (! (o instanceof Int)) {
      return false;
    }
    Int i = (Int) o;
    return this.n == i.n;
  }
}

/** A IntInt is a Int with an additional integer value. */
class IntInt extends Int {
  private int x;
  public IntInt(int n, int x) {
    super(n);
    this.x = x;
  }
  public boolean equals(Object o) {
    if (! (o instanceof Int)) {
      return false;
    }
    if (! (o instanceof IntInt)) {
      return super.equals(o);
    }
    IntInt ii = (IntInt) o;
    return super.equals(ii) && this.x == ii.x;
  }
}
```