Prelim 1, Solution

CS 2110, 12 March 2019, 5:30 PM

	1	2	3	4	5	6	Total
Question	Name	Short answer	OO	Recursion	Loop invariants	Exception handling	
Max	1	32	23	19	13	12	100
Score							
Grader							

The exam is closed book and closed notes. Do not begin until instructed.

You have **90 minutes**. Good luck!

Write your name and Cornell **NetID**, **legibly**, at the top of the first page, and your Cornell ID Number (7 digits) at the top of pages 2-8! There are 6 questions on 8 numbered pages, front and back. Check that you have all the pages. When you hand in your exam, make sure your pages are still stapled together. If not, please use our stapler to reattach all your pages!

We have scrap paper available. If you do a lot of crossing out and rewriting, you might want to write code on scrap paper first and then copy it to the exam so that we can make sense of what you handed in.

Write your answers in the space provided. Ambiguous answers will be considered incorrect. You should be able to fit your answers easily into the space provided.

In some places, we have abbreviated or condensed code to reduce the number of pages that must be printed for the exam. In others, code has been obfuscated to make the problem more difficult. This does not mean that it's good style.

Academic Integrity Statement: I pledge that I have neither given nor received any unauthorized aid on this exam. I will not talk about the exam with anyone in this course who has not yet taken prelim 1.

_____

(signature)

1. Name (1 point)

Write your name and NetID, **legibly**, at the top of page 1. Write your Student ID Number (the 7 digits on your student ID) at the top of pages 2-8 (so each page has identification).

2. Short Answer (32 points)

(a) 6 points. Below are three expressions. To the right of each, write its value.

1. `(char) ('b'+ 2)` 'd' char is numerical type.
2. `new Double(3.14) == new Double(3.14)` false These are two separate objects.
3. `(int) (4.8/2) == (int) (4.8/2)` true

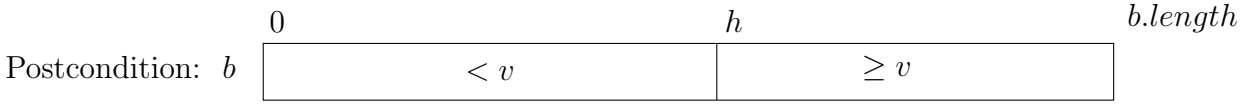
(b) 8 points. Circle T or F in the table below.

(a)	T	F	`if (1) { return "Correct"; }` is valid Java. false 1 is not of type boolean
(b)	T	F	Quicksort is not stable. true partition algorithm is inherently unstable.
(c)	T	F	The tightest worst-case bound for Mergesort is $O(n^2)$ because its depth of recursion may be proportional to the length n of the array. false The depth of recursion of Mergesort is $O(log\ n)$.
(d)	T	F	Space for a local variable declared within a loop body is allocated each time the loop body is executed and deallocated when the loop body finishes. false The four steps in executing a method call tell us differently.
(e)	T	F	Two versions of an overloaded method can have different numbers of parameters. true For example, String has two substring functions, one with 1 parameter and the other with two.
(f)	T	F	Even though `Comparable` is an interface, the following declaration is legal: `Comparable c;` . true This is just a declaration of a variables with a type.
(g)	T	F	Even though `Comparable` is an interface, the following statement is legal: `Comparable c= new Comparable();` . false The new-expression is illegal since Comparable is an interface.
(h)	T	F	This declaration overrides method `toString()`: `public String toString(int x) { return "" + x; }` . false To override toString(), it should have no parameters.

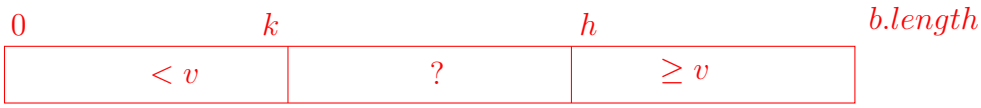
(c) 3 points. Write the 3-step algorithm for evaluating the new-expression `new XYZ(2.1)` .

1. Create a new object of class XYZ
2. Execute the constructor call new XYZ(2.1).
3. Use as value of the new-expression the name of (pointer to) the new object - the stuff that was written in the tab of the new object.

(d) 3 points. **Binary search.** The precondition of binary search is that array `b` is sorted. The postcondition is given below. If `v` is in `b` it indicates that the leftmost occurrence of `v` will be found. Complete the loop invariant, which appears below the postcondition. There is no need to mention that `b` is sorted, since that is always true.



Answer:



(e) 6 points. Complexity

1. Suppose a method calls a procedure that requires $O(n\ log\ n)$ operations once and then calls another procedure $n/2$ times, and this second procedure requires n operations. Below, circle the tightest (smallest) asymptotic time complexity of this method.

$O(n)$ $O(n\ log\ n)$ $O(n^2)$

$O(n^2)$ The total operations executed in all calls of the second procedure is $n * n/2$.

2. To the right of the code below, write the tightest (smallest) asymptotic time complexity (in terms of n) of the code. O(n). The inner loop always executes 5 iterations; it's constant time.

```
int s= 0;
for (int h= 0; h < n; h= h+1) {
    for (int j= 0; j < 5; j= j+1) {
      s= s + h*j;
    }
}
```

(f) 6 points. Testing. Below is the signature for function findClosest.

```
/** Return the value in b that is closest to zero.
 * Note: -2 is closer to zero than 3.
 * If several array values are the same distance from zero, return
 * the one with smallest index.
 * If b is null or empty, return Integer.MAX_VALUE.  */
public static int findClosest(int[] b)
```

Write six (6) distinct test cases in the space below. We don't need formal `assertEquals` calls. Instead, say what is in `b` (or give its values as a list).

One should be convinced the function works as specified if it passes all six of these test cases. Testing nearly identical cases 6 times does not count.

We're looking for some combination of

1. b is null.
2. b is empty.
3. b contains 0.
4. b contains multiple values the same distance from 0, e.g 2, -2.
5. b contains all positive values.
6. b contains all negative values.
7. b contains a mix of positive and negative values.

3. Object-Oriented Programming (23 points)

Below and on the next page are three classes, `Restaurant`, `FancyRestaurant`, and `Review`. Unnecessary parts of classes are omitted. There is no need for assert statements for preconditions.

(a) 3 points Implement `Restaurant`'s constructor.

(b) 3 points Implement `Restaurant`'s method `compareTo`.

(c) 7 points Implement `Restaurant`'s method `equals`.

(d) 6 points Implement `FancyRestaurant`'s constructor.

(e) 4 points Implement `FancyRestaurant`'s method `equals`.

```
/** An instance contains the name of a restaurant and reviews of it. */
class Restaurant implements Comparable<Restaurant> {
  private String name;                // Name of the restaurant (not null)
  private ArrayList<Review> reviews; // list of reviews of this restaurant

  /** Constructor: a restaurant with name name and no reviews.
    * Precondition: name is not null. */
  public Restaurant(String name) { // TODO: Part a
    this.name= name;
    reviews= new ArrayList<>();
    // Not assigning to reviews is wrong. null is not an empty ArrayList.
  }

  /** Return this restaurant's average rating (rounded to nearest integer). */
  public int avRating() { // ... implementation omitted ... }

  /** Compare average ratings (rounded to the nearest integer)
    * as per specification in interface Comparable. */
  @Override public int compareTo(Restaurant r) {  // TODO: Part b
        return avRating() - r.avRating();
        // you can use if-statements or conditional expression instead,
           and -1,0, or 1 can be returned.
  }

  /** Return true iff this Restaurant and ob are of the same class and
      have the same name. */
  @Override public boolean equals(Object ob) { // TODO: Part c
      if (ob == null  || getClass() != ob.getClass()) return false;
      Restaurant r= (Restaurant) ob;
      return name.equals(r.name);



  }
}
```

```
/** FancyRestaurant is fancy: It has tables and reservations for them. */
class FancyRestaurant extends Restaurant {
  int numTables;                   // Number of tables
  ArrayList<String> reservations; // List of reservations made (not null)

  /** Constr.: instance with name n, number of tables tables, no reservations.
    * Precondition: n is not null */
  public FancyRestaurant(String n, int tables) { // TODO: Part d
      super(n);
      numTables= tables;
      reservations= new ArrayList<>();
      // Not assigning an empty list to reservations is wrong;
      // null is not an empty list
  }

  /** Return true iff this object and ob are of the same class, they have
    * the same name, and they have the same number of tables. */
  public boolean equals(Object ob) { // TODO: Part e
      if (!super.equals(ob)) return false;
      FancyRestaurant fr= (FancyRestaurant) ob;
      return numTables == fr.numTables;
      // This must start off with the call on super.equals. Just as in
      // in a subclass constructor, the superclass fields are filled in
      // first, in the OO way, check the superclass equals first.
      // Checking getClasses in this method is redundant since they
      // are checked in super.equals.
  }
}

/** A review consists of a written review and a rating. */
class Review {
  // We omit the implementation of this class.
}
```

4. Recursion (19 Points)

(a) 4 points The Hofstadter F and M sequences are defined recursively as follows:

```
public static int F(int n) {
  if (n == 0) return 1;
  return n - M(F(n - 1));
}
```

```
public static int M(int n) {
  if (n == 0) return 0;
  return n - F(M(n - 1));
}
```

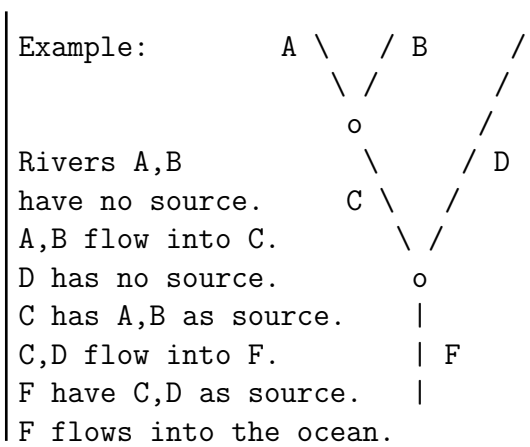
Write the calls made in evaluating the call `F(1)` in the order they are called, starting with `F(1)`.

`F(1), F(0), M(1), M(0), F(0)`

Class River, below, maintains information about rivers. A river may have no source or one or two other rivers, which feed into it. A river feeds into another river or goes directly into the ocean.

(b) 7 points. Complete the body of function distanceToOcean(). Use recursion; do not use loops.

(c) 8 points. Complete the body of function isPolluted(). Use recursion; do not use loops.



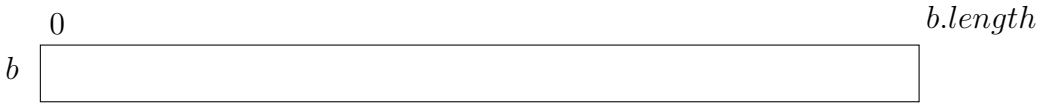

```
public class River {
  private River leftSource;   // left source of river, null if none
  private River rightSource;  // right source of river, null if none
  private River feeds; // river into which this one feeds (null if it goes to ocean)
  private double len;         // length of this river in kilometers.
  private boolean hasFactory; // true if a factory is polluting this river.

  /** Return the length (distance) from the start of this river to the ocean */
  public double distanceToOcean() {
    if (feeds == null) return len;
    return len + feeds.distanceToOcean();
  }

  /** Return true if this river is polluted. A river is polluted if a
    *  factory is polluting it or either of its sources are polluted. */
  public boolean isPolluted() {
    if (hasFactory) return true;
    if (leftSource != null && leftSource.isPolluted()) return true;
    return rightSource != null && rightSource.isPolluted();
    // It's alright to start off with the base case like this:
    //   if leftSource == null  &&  rightSource == null) return hasFactory;
    // Our solution is shorter.
  }
}
```

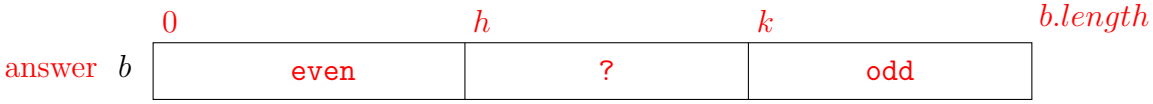
5. Loop Invariants (13 points)

(a) 3 points Consider the assertion

`b[0..h-1] are even  && h ≤ k  && b[k...b.length-1] are odd`

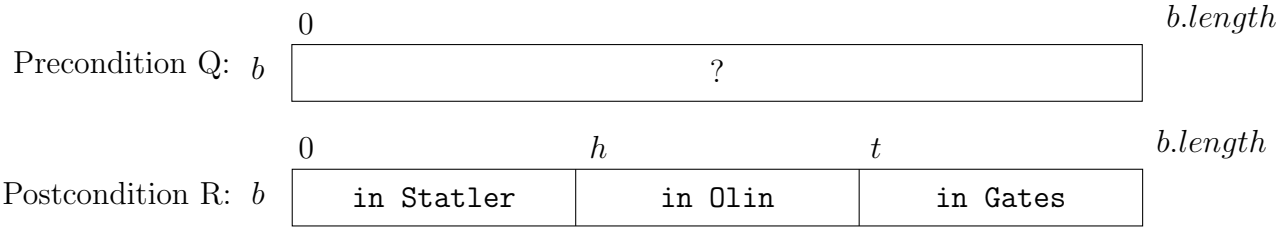
Draw it as an array diagram below:





When the loop terminates, h and k are equal.
Don't draw h or k directly above a line; that is ambiguous.
The section b[h..k] must be there because of the term h ¡= k.
It doesn't matter what you put in it —? or unknown or leave it blank.

(b) 10 points Below are the precondition, postcondition, and invariant of a loop that sorts an array *b* of Students based on the hall in which they are taking a prelim (Statler, Olin, and Gates):



1. Write the initialization
   code that truthifies P: *h*= 0; *m*= 0; *t*= *b.length*;

2. Write a while-loop condition that makes
   the loop terminate when R is true: `m != t` or use this alternative: `m < t`

3. Write the repetend. Use `swap(b, i, j)` to swap `b[i]` and `b[j]`. Instances of class Student have three functions `inStatler()` (true if the student is in Statler Hall), `inOlin()` (true if the student is in Olin), and `inGates()` (true if the student is in Gates).

```
if (b[m].inStatler()) { swap(b, m, h);  h= h+1;  m= m+1;  }
else if (b[m].inGates()) { t= t-1;  swap(b, m, t); }
else m= m+1;
// When we look at the repetend, we do not take into account what the
// initialization is. We look only at whether this Hoare triple is true:
//    {invariant AND m < t}   repetend {invariant}
// and whether the repetend makes progress toward termination.
// Don't look at your initialization when writing the repetend.
```

6. Exception handling (12 Points)

In method `foo` to the right, `S1`, `S2`, `S3`, `S4`, `S5`, and `S6` are statements. Below, method `m` calls `foo`.

```
static void m() {
    ...
    foo();
    ...
}
```

Below are four situations that could happen when the call on `foo` in method `m` is executed. Answer the question for each of these situations.

```
public static void foo()  {
    S1;
    try { S2 }
    catch (RuntimeException e) { S3 }
    catch (Error e) { S4 }
    catch (Throwable e) { S5 }
    S6;
}
```

(a) 3 points Suppose `S1` throws an `Error`. Is it caught by the second catch clause, and if not, what happens to that `Error` object?
It is not caught because S1 is not within the try-block. It is thrown out to the call of foo in m.

(b) 3 points Suppose `S2` throws an `Error`. Is it caught by the second catch clause, and if not, what happens to that `Error` object?
Yes, it is caught by the second catch clause.

(c) 3 points Suppose `S4` throws a `Throwable`. Is it caught by the third catch clause, and if not, what happens to that `Throwable` object?
It is not caught because S4 is not within the try-block. It is thrown out to the call of foo in m.

(d) 3 points Suppose `S4` is executed and it does not throw anything. What is executed next?
S6 is executed next.