Prelim 1, Solution

CS 2110, 27 September 2018, 5:30 PM

	1	2	3	4	5	6	Total
Question	Name	Short answer	Exception handling	Recursion	OO	Loop invariants	
Max	1	34	8	16	31	10	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 **every** page! There are 6 questions on 10 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 **every** page of this exam.

2. Short Answer (34 points)

(a) 6 points. Each of the expressions below either (1) does not compile, (2) throws an exception at runtime, or (3) can be evaluated to produce a value. To the right of each one, state which case it is, and for (3), write down its value.

1. `(char)('0'+ 7)` '7' char is numerical type.
2. `null + 18` Error Null will not automatically be cast to an int to be added.
3. `'E' + "zra" + 65` Ezra65 'E' and 65 will be cast to String to perform a legal catenation.
4. `"A.D.White".substring(3).charAt(3)` `'i'`. `"A.D.White".substring(3)` evaluates to `".White"`, and `charAt(3)` then produces `'i'`.
5. For a variable `s` declared as `String s = new String();`,
   `s == s.substring(0);` true `s.substring(0)` is the same as s, and a ptr to s is the value; no new string will be created.
6. `10/4.0` 2.5 `10` will be cast to a double for division. No truncation is done.

(b) 5 points. Circle T or F in the table below.

(a)	T	F	Non-static methods can access static fields. true The inside-out rule tells you that this is possible.
(b)	T	F	The class invariant must remain true throughout execution of all methods. False Class invariants must only be true at the beginning and end of each method call.
(c)	T	F	In a subclass constructor, you must explicitly call the superclass constructor. False You can use this(...), and if you leave that out, Java inserts one for you.
(d)	T	F	At runtime, downward type casts will be done automatically if necessary. False Upward casts will be done automatically, not downward.
(e)	T	F	Given a class `A` that implements the `Comparable` interface, if the programmer doesn't define a `compareTo` method, Java will use the memory locations to compare two objects of `A`. False The programmer must define a compareTo method since `A` implements `Comparable`, and if not there will be a compilation error.

(c) 7 points. Use classes Animal and Cow below to answer the following questions.

```
public abstract class Animal {
    public Animal() {
        System.out.println("I am an animal!");
    }

    public void makeSound() {
        System.out.println("@#&*%!");
    }
}

public class Cow extends Animal {
    String farm;

    public Cow(String farm) {
        super();
        this.farm= farm;
        System.out.println("I am a cow from " + farm + "!");
    }

    public void makeSound() {
        System.out.println("Moo!");
    }
}
```

Write down what will be printed in the terminal when the following code is executed.

```
Animal emma= new Cow("Cornell Dairy Barn");
emma.makeSound();
```

```
I am an animal!
I am a cow from Cornell Dairy Barn!
Moo!
```

(d) 6 points. Implement function `isEqual` whose specification is given below. Do not create any new String or array objects.

```
/** Return true if the list of characters in array c (in-order) is the same
as the case-sensitive, ordered sequence of characters in s.
Throw an IllegalArgumentException if c or s is null.

    Examples:
        char[] array1=  {'g', 'r', 'i', 'e', 's'};
        char[] array2=  {'r', 'g', 'i', 's', 'e'};
        isEqual(array1, "gries") is true
        isEqual(array2, "Gries") is false
        isEqual(array2, "gries") is false
*/
public static boolean isEqual(char[] c, String s) {


  if (c == null || s == null) {
    throw new IllegalArgumentException();
  }
  if (c.length != s.length()) return false;
  for (int i= 0; i < c.length; i++) {
    if (s.charAt(i) != c[i]) return false;
  }
  return true;

}
```

(e) 6 points. Write 4 test cases based upon the specification of `isEquals` (black box testing). Do not write assertEquals(...) and all that. Just give the two arguments of a call on `isEquals` for the test case. Answer do not have to compile, could be partly in English.

Here are 6 test cases, based on the specification:

1. c is null and s is not null
2. s is null and c is not null
3. Lengths of c and s are different
4. At some index i, c[i] and s[i] are different
5. c and s contain the same nonempty list of chars
6. c is {'A'} and s is "a" (check case sensitivity)

(f) 4 points. Write the four steps in executing the procedure call `C(int k);` .

1. Push a frame for the call onto the call stack.
2. Assign the value of argument k to the parameter (in the frame for the call).
3. Execute the method body.
4. Pop the frame for the call from the call stack. (Since this is for a procedure call, there is no result value to push onto the call stack.)

3. Exception handling (8 Points)

```
public static void foo(int a, int b) {
    System.out.println("1 ");
    int res= a / (b * 5);
    try {
      System.out.println("2");
      if (b == 0 || b == -1)  throw new IllegalArgumentException();
      System.out.println("3");
      a= b / (b-1);
      System.out.println("4");
    }

    catch (ArithmeticException e) {
      System.out.println("5");
      if (b == 1)  throw new RuntimeException();
      System.out.println("6");
    }

    catch (Exception e) {
      System.out.println("7");
    }

    System.out.println("8");
  }
```

What would be the output to the console for the following three statements?

1. foo(-1,-1);
   These numbers are printed on separate lines: 1, 2, 7, 8
2. foo(0,0);
   This number and exception are printed on separate lines: 1, ArithmeticException
3. foo(1,1);
   These numbers and exception are printed on separate lines: 1, 2, 3, 5, RuntimeException

4. Recursion (16 Points)

(a) 8 points Consider the following recursive function:

```
public static void hailstone(int n) {
  if (n > 0) {
      System.out.println(n);
      if (n != 1) {
          if (n % 2 == 0) // n is even, halve it
              hailstone(n / 2);
          else // n is odd, times by 3 and add 1
              hailstone(3*n + 1);
      }
  }
}
```

Execute the following calls and write down what is printed for each, stop after executing 4 println statements.

hailstone(1):	hailstone(2):	hailstone(5):
1	2	5
	1	16
		8
		4

(b) 8 points Write the body of recursive procedure `comify`. **You must use recursion; do not use a loop!** You can use function `p`, given below.

```
/** Return m as a string with commas every 3 digits, as done in the U.S.
  * Precondition: m >= 0.
  * E.g. for m = 3152463, return the string "3,152,463".  */
public static String comify(int m) {
    if (m < 1000) return "" + m;
    return comify(m/1000) + "," + p(m % 1000);
}

/** = n but with leading 0's, if necessary, to have at least 3 chars.
      Precondition: 0 <= n */
public static String p(int n) {
   if (n < 10) return "00" + n;
   if (n < 100) return "0" + n;
   return "" + n;
}
```

5. Object-Oriented Programming (31 points)

Below is abstract class `Chef`, which is used throughout this problem:

```
public abstract class Chef implements Comparable                    {
   private String name; // Name of chef

   /** Constructor:  Chef with name n.  */
   public Chef(String n) {name= n;}

   /** Return a representation of this Chef ---their name.  */
   public String toString() {return name;}

   /** Return true if chef is busy and false otherwise */
   public boolean isOccupied() {... Assume implemented ...}

   public abstract boolean canMakeFood();

   public abstract void makeFood();

   /** Return a negative int, 0, or positive int depending on whether
     * this chef's name comes before, is the same as, or comes after
     * ob's name, in dictionary order.  Throw a ClassCastException
     * if ob is not a Chef.  */
   public int compareTo(Object ob) {
      Chef obs= (Chef) ob;
      return name.compareTo(obs.name);
   }

}
```

(a) 5 points Recall that interface `Comparable` declares abstract function `compareTo(Object ob)`. This function is declared in class `Chef`. Complete its body and make one other change in class `Chef` so that it implements interface `Comparable`.

Here are two hints. (1) Casting `ob` to `Chef` throws a `ClassCastException` if `ob` is not a `Chef`. (2) Class `String` implements function `compareTo(String)` with essentially the same specification.

(b) 12 pts Class `BakingChef` (below) extends `Chef`.

(1) Complete the body of the constructor.

(2) Complete the return statement in function `canMakeFood`, assuming the two boolean variables are assigned properly (you do not have to do this).

(3) Complete the body of procedure `makeFood`.

```
public class BakingChef extends Chef {
   private String[] inventory; // list of ingredients

   /** Constructor:  an instance with name n and inventory inventory.*/
   public BakingChef(String n, String[] inventory) {
      super(n);
      this.inventory= inventory;
   }

  /** Return true if (1) the BakingChef is not busy and */
    * (2) the inventory has both eggs and flour.  */
  public boolean canMakeFood() {
      boolean hasEggs= ... code to set true if eggs are in inventory;
      boolean hasFlour= ... code to set true if flour in inventory;
      return !isOccupied() && hasEggs && hasFlour;
  }

  /** If able to make food, print "baking"; otherwise throw
    * a RuntimeException with message "cannot bake".  */
  public void makeFood() {
      if (canMakeFood()) System.out.println("baking");
      else throw new RuntimeException("cannot bake");
  }
```

(c) 5 points The following function `toString` is to be added to class `BakingChef`. Complete its body.

```
/** Return name of chef and size of inventory.  */
public String toString() {
   return super.toString() + " " + inventory.length;
}
```

(d) 9 points `CakeChef` (below) is a subclass of `BakingChef`. It has three syntax errors, so it doesn't compile. Identify the three syntax errors.

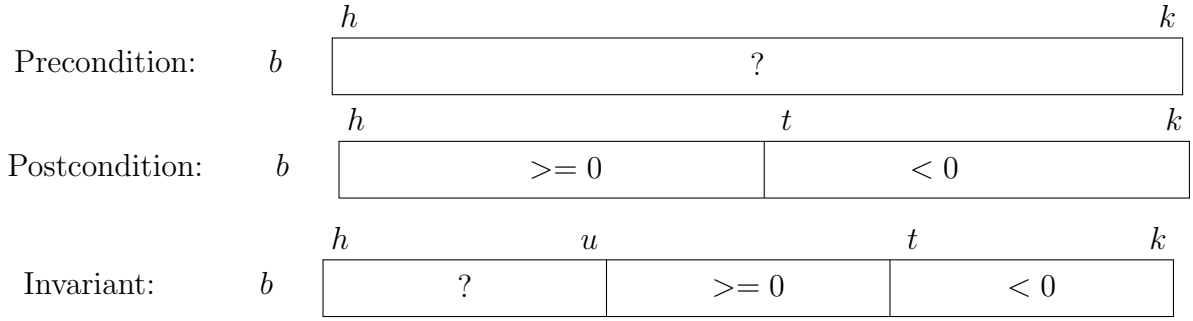
```
public class CakeChef extends BakingChef {
 /** chef with name n and inventory inv.  */
 public CakeChef(String n, String[] inv) {
    // Error:  no body, so super(); is inserted and doesn't compile because the
superclass does not contain a constructor with no parameters.
 }

 @Override // Error:  this is overloading, not overriding, so it doesn't compile.
 public String toString(String s) {
    return s;
 }

 public static String getWho() {
    return toString("who"); //Error:  static method calls non-static method.
 }
}
```

6. Loop Invariants (10 points)

Below are the precondition, postcondition, and invariant of a loop that swaps the negative values in $b[h..k]$ to the end of $b[h..k]$. Note that h is not necessarily 0, k is not necessarily $b.length-1$, and both h and k should not be changed. You do not have to be concerned with declaring variables.



(a) 2 pts Given that the Precondition is true, write an initialization that truthifies the Invariant:
u= k; t= k+1;

(b) 3 points Write a while-loop condition. Make sure that if the while-loop condition is false, the Invariant implies the Postcondition. h <= u or h != u+1 or u != h-1

(c) 5 points Given that segment $b[h..u]$ is not empty, write a loop body that keeps the Invariant true and makes progress toward termination (by decreasing the size of $b[h..u]$).
Note: Use a procedure call $swap(c, i, j)$ to swap array elements $c[i]$ and $c[j]$.

```
if (b[u] >= 0)  u= u-1
else {t= t-1;  swap(b, u, t);  u= u-1;}
```

OR

```
if (b[u] < 0) {t= t-1; swap(b, u, t)};
u= u-1;
```