NAME______________________ RCS ID_________________

Your RCS ID is the First Part of Your RPI Email

It is not your RIN! Put Your name and RCS ID on each page.

Exam 2
CSCI 2600 Principles of Software
Nov. 8, 2018

- READ THROUGH THE ENTIRE EXAM BEFORE STARTING TO WORK.
- YOU ARE ALLOWED A SINGLE DOUBLE-SIDED SHEET OF NOTES.
- NO OTHER MATERIAL IS ALLOWED.

PLEASE PUT YOUR NAME AND RCS ID ON EACH PAGE

This exam is worth 100 points.

Make sure you have 12 pages counting this one. If you need more room for an answer than is provided, please use the back of the page and indicate that you have done so. If you re-do a question, please make clear what is your final answer.

Be clear and brief in your explanations—rambling and lengthy answers will be penalized. All questions have short answers. Please write neatly. If we cannot read your answer, we cannot grade it.

Assume no overflow or underflow conditions occur in any of the code.

The following is for the use by the graders

1.	__________/24
2.	__________/20
3.	__________/12
4.	__________/14
5.	__________/14
6.	__________/8
7.	__________/8

TOTAL:___________/100

Name: ___________________________ RCS ID ________________________________

Question 1. (24 points)

Our friend Willie Wazoo has created the code below to implement a finite string bag (FSB). A FiniteStringBag is a set that allows duplicate values, also called a multiset. The representation uses a fixed length array to hold the elements and has an associated integer variable to record the size of the bag (the number of elements being used in the array).

```
class FiniteStringBag {
      // Rep: items is a fixed length array
      // items[0..size-1] contains Strings
      // size <= items.length
      // Entries cannot be null. There may be multiple copies
      // of the same string in the FiniteStringBag (FSB).
      private String[] items;
      private int size;

      //Construct new FSB with given capacity.
      // Requires: capacity > 0
      public FiniteStringBag(int capacity) {
            this.items = new String[capacity];
            this.size = 0;
      }

      /** Return capacity of this FSB */
      public int capacity() { return items.length; }
      /** Return current size of this FSB*/
      public int size() { return size; }

      /** Return item at position i of this FSB */
      public String get(int i) {
            if(i >= size)
                  throw new IndexOutOfBoundsException();
            return items[i];
      }
      public void add(String s) {
            if(size == items.length)
                  throw new BufferOverflowException();
            items[size] = s;
            size = size + 1;
      }
      /** Return whether s is located in this FSB */
      public boolean contains(String s) {
            if (size == 0) return false;
            for(int i = 0; i < size; i++) {
                  if(items[i].equals(s))
                        return true;
            return false;
      }
      // delete strings with length > n
      public void deleteLongStrings(int n) {
            int k = 0;
            while (k < size) {
                  if (items[k].length() > n) {
                        items[k] = items[size-1];
                        size = size-1;
                  } else {
                        k = k + 1;
                  }
            }
      }

      // additional methods to be added later....
```

a) (4 points)
Give a suitable abstraction function (AF) for this class relating the representation to the abstract value of a finite string bag.

b) (5 points)

`FiniteStringBag` lacks a `checkRep()` method. Write a `checkRep()` method for this class. The method should check the representation as described in the comments at the start of the class definition.

c) (9 points)
Describe three separate, distinct “black box” tests for the deleteLongStrings method. You don’t need to write Java or Junit code. Just give a brief clear description of the test.

d) (2 points)

Are there any potential problem from representation exposure with the FiniteStringBag as it written above? Why or why not? Be brief.

e) (4 points)

We would like to add an observer method to FiniteStringBag. Willie proposes the following method

```
// return the current strings in this FSB to the caller
public String[] getItems() {
     return items;
}
```

Willie's colleague Ima Hacker points out that there are two problems with this method. What are they?

Question 2. (20 points) TRUE/FALSE

a) (TRUE/FALSE) It is always possible to cover all def-use pairs in a function.

b) (TRUE/FALSE) When testing machine learning (ML) routines, we typically use white-box tests.

c) (TRUE/FALSE) If A is a true subtype of B, then an instance of B can be safely substituted for A.

d) (TRUE/FALSE) In Java, if two objects are not equal according to the `equals(Object)` method, the `hashCode` methods of each should produce different int values.

e) (TRUE/FALSE) A test suite that has 100% statement coverage necessarily covers all def-use paths.

f) (TRUE/FALSE) A method `String t(Integer)` is a true function subtype of method `Object t(Number)` Note: `Integer` is a subclass of `Number`.

g) (TRUE/FALSE) In Java an overloaded method must have the same number and type of arguments as the method in the superclass.

h) (TRUE/FALSE) If `equals(Object)` is reflexive and symmetric, it is guaranteed to be transitive.

i) (TRUE/FALSE) In Java, checked exceptions must either be declared in a throws clause of the calling method or caught by the calling method via a try/catch block.

j) (TRUE/FALSE) In Java, unchecked exceptions are used to convey special results to the calling method.

Name: ______________________________ RCS ID ___________________________________

Question 3 (12 points)

```
class A {
        void m(A a) { System.out.println("AA"); }
        void m(B a) { System.out.println("AB"); }
        void m(C a) { System.out.println("AC"); }
}
class B extends A {
        void m(A a) { System.out.println("BA"); }
        void m(B a) { System.out.println("BB"); }
        void m(C a) { System.out.println("BC"); }
}
class C extends B {
        void m(A a) { System.out.println("CA"); }
        void m(B a) { System.out.println("CB"); }
        void m(C a) { System.out.println("CC"); }
}
public class CallTestDemo {

        public static void main(String[] args) {
                A a1 = new A();
                A a2 = new B();
                B a3 = new C();
                A b1 = new B();
                B c1 = new C();
        …
        }
}
```

What is printed by each of the following calls?

a1.m(a2);

a2.m(b1);

a2.m(c1);

b1.m(a1);

c1.m(b1);

a3.m(b1);

Name: ___________________________ RCS ID ________________________________

Question 4 (14 points)

Please keep your answers short and clear.

a) (2 points)
You are writing a library that allows client code to read and write compressed data for storing in files. You've written several classes that implement methods of a Compressed File interface. These classes use different algorithms with different characteristics. Rather than requiring the client to choose among them, you want the client to be able simply to request a Compressed File object and be given an instance of the best available implementation for the particular kind of file. What type of design pattern that we have covered would be most appropriate?

b) (2 points)
What is a limitation of Java constructors that the factory pattern overcomes?

c) (2 points)
Would it be appropriate to use the Interning design pattern with `RatNum` objects from Homework 3? If so, describe a situation when it might be useful. If not, justify why not.

d) (2 points)
Would it be appropriate to use the Interning design pattern with a large number of `Graph` objects from Homework 4? If so, describe a situation when it might be useful. If not, justify why not.

e) (2 points)
Your program can send content to a printer. You want to ensure that different parts of the program doesn't command the printer to print output belonging to multiple documents simultaneously, mixed together. What type of design pattern that we have covered would be most appropriate?

f) (2 points)
Suppose we have a class A with a method m:

```
    class A {
        public T m(S x) { ... }
    }
```

Now suppose we create a class B that is a subclass of A with its own method m that is supposed to override the one from class A:

```
    class B extends A {
        public T1 m(S1 x) { ... }
    }
```

If we want class B to be a true subtype of class A, what are the possible subtype/supertype relationships between m's types T and S in A and types T1 and S1 in B?

g) (2 points)
We want to refactor the following method so that it uses generic types.

```
// effects: dst is a copy of src without duplicates.
// modifies: dst. src is not modified.
static void removeDups(Collection src, Collection dst);
```

Fill in the blanks in the following signature with either `extends` or `super`.

```
static void removeDups(Collection<? __________________________ T> src,

                       Collection<? _________________________  T> dst);
```

Name: ______________________________ RCS ID ____________________________________

Question 5 (14 points)

Consider the following class hierarchies:

```
class Mammal {}
class Cow extends Mammal {}
class Horse extends Mammal {}
class ToyHorse extends Horse {}
```

and the following variables:

```
Object o; Mammal m; Cow c;  Horse h; ToyHorse t;
List<? extends Mammal> lem;
List<? extends Horse> leh;
List<? super Horse> lsh;
```

For each of the following, circle OK if the statement has the correct Java types and will compile, otherwise circle ERROR.

```
a) lem.add(h);          OK        ERROR

b) lsh.add(t);          OK        ERROR

c) lsh.add(o);          OK        ERROR

d) lem.add(null);       OK        ERROR

e) h = lsh.get(1);      OK        ERROR

f) m = leh.get(1);      OK        ERROR

g) o = lsh.get(1);      OK        ERROR
```

Name: ___________________________ RCS ID ________________________________

Question 6 (8 points)

An equivalence relation should have the properties of being *reflexive*, *symmetric*, and *transitive*. Consider the following code:

```
class ConstantInt {
      private Integer val;
      public ConstantInt(int v) {
            this.val = v;
      }
// other methods …
}
```

```
      ConstantInt a = new ConstantInt(1);
      Object b = new ConstantInt(1);
      ConstantInt c = new ConstantInt(1);

       System.out.println(a.equals(a));
       System.out.println(a.equals(b));    System.out.println(b.equals(c));
       System.out.println(b.equals(a));    System.out.println(c.equals(a));
```

We wish to override `equals` for `ConstantInt`. Indicate which of the following implementations of `equals` has all three properties by selecting VALID or NOT VALID. That is, to be valid all the above output statements should print *true*.

```
public boolean equals(ConstantInt x) { return this == x; }
```

VALID NOT VALID

```
public boolean equals(ConstantInt x) { return this.val == x.val; }
```

VALID NOT VALID

```
public boolean equals(Object x) { return this.val.equals(x); }
```

VALID NOT VALID

```
public boolean equals(Object x) {
        if(this == x)
          return true;
        if(!(x instanceof ConstantInt))
          return false;
        ConstantInt ci = (ConstantInt)x;
        return this.val == ci.val;
}
```

VALID NOT VALID

Name: ___________________________ RCS ID ________________________________

Question 7 (8 points)

Consider the following method:

```
int m(boolean a, boolean b, boolean c, boolean d) {
      int ans = 1;
      if (a) {
            ans = 2;
      } else if (b) {
            ans = 3;
      } else if (c) {
            if(d) {
                  ans = 4;
            }
      }
      return ans;
}
```

a) What number of tests do you need to exhaustively test this method?

b) What is the minimum number of tests needed to achieve full statement coverage for this method?

c) What is the minimum number of tests needed to achieve full branch coverage?

d) Which of the questions above can be answered while using only a black-box testing methodology?