Com S 362
Fall 2004

Name: _____

Object-Oriented Analysis and Design

Exam 1 on Abstract Data Types and Java

This test has 3 questions and pages numbered 1 through 15.

Reminders

This test is open book and notes. However, it is to be done individually and you are not to exchange or share materials with other students during the test.

If you need more space, use the back of a page. Note when you do that on the front.

This test is timed. We will not grade your test if you try to take more than the time allowed. Therefore, before you begin, please take a moment to look over the entire test so that you can budget your time.

For diagrams and programs, clarity is important; if your diagrams or programs are sloppy and hard to read, you will lose points. Correct syntax also makes some difference.

---

1. (10 points) Consider a data type List, with public methods `isEmpty()` and `head()`. The method `isEmpty()` returns true when the list has no elements. The method `head()` returns the first element of a list. The method `head()`'s precondition says that, in a call such as `myList.head()`, `myList` cannot be empty; that is, `myList` must have some elements for this call to be considered proper. When called improperly, that is when called on an empty list, the `head()` method should throw an exception. Should this exception be a checked exception? (a) Answer "yes" or "no" and (b) explain why, briefly.

2. This is a problem about simple applications, file I/O, and exceptions in Java. The program consists of 2 classes, found below, that merge sorted input files. The details are in the program's comments. But here are some examples.

   Suppose we have the following in the file `input1.txt`.

```
a first line
before you look
cede the ground
determine where you are
enough!
```

   And suppose that `emptyfile.txt` has no lines of text in it. Then both of the following commands

```
java cs362exams.MergeMain input1.txt emptyfile.txt output.txt
java cs362exams.MergeMain emptyfile.txt input1.txt output.txt
```

   put the contents of `input1.txt` in the file `output.txt`.

   Suppose we have the following in the file `input2.txt`.

```
a first line
beware the sorting
can go upward
```

   Then the command

```
java cs362exams.MergeMain input1.txt input2.txt output.txt
```

   produces the following in output.txt:

```
a first line
a first line
before you look
beware the sorting
can go upward
cede the ground
determine where you are
enough!
```

Please write your code by filling in the empty methods in the code below.

(a) (10 points) In the following class, fill in the body of the method "`main`". (There is space for your answer, and a private helping method, `checkArguments`, on the next page.)

```
package cs362exams;
import java.io.IOException;

/**
 * Merge the lines from the first two files and put the output in the third
 * file. The files have their names given as command line arguments; this
 * application exits with exit code 2 if three arguments were not given. The
 * contents of the first two files are assumed to be lines that are sorted in
 * ascending order. These lines, retaining all duplicates, are written to the
 * third file. The third file might exist already, and is created if it does not
 * already exist. Normally, this application exits with exit code 0.
 * But if there was any I/O error, then an error message is printed to
 * System.err, and the application exits with exit code 1.
 */
public class MergeMain {

    /** Run this application.  This method is responsible for calling
     * the doMerge() method of the class Merge.  This method is also
     * responsible for catching any IOExceptions that doMerge() may throw;
     * if it catches any such exceptions, then it should print
     * their message to System.err, preceeded by
     *    "MergeMain: I/O error: "
     * and followed by a newline, and then exit with exit code 1.
     * This method must also call the
     * checkArguments method and exit with a exit code 2 if
     * there are not three arguments in argv.
     * @param argv contains the three file names;
     *        the first two are names of input files,
     *        and the third is the name of the output file. */
    /*@ requires argv != null && \nonnullelements(argv);
      @ ensures (* if the exit code is 0,
      @            then the lines of the sorted input files named by
      @            argv[0] and argv[1] are merged into the file
      @            named by argv[2]. *);
      @ ensures (* if the exit code is 1, then an I/O error occurred,
      @            and details were written to System.err. *);
      @ ensures (* if the exit code is 2, then the length of argv is not 3 *);
      @*/
    // ... put code for public static void main(String[] argv) on next page...
```

```
    public static void main(String[] argv) {









    }

    /** Check for proper usage of arguments,
     * and issue appropriate error messages.
     * @return false, if there not 3 arguments, true otherwise. */
    //@ ensures !\result || \old(argv.length == 3);
    private static boolean checkArguments(String[] argv) {
        if (argv.length != 3) {
            System.err.println("MergeMain: expecting 3 arguments");
            return false;
        } else {
            return true;
        }
    }
}
```

(b) (10 points) In the following class, "`Merge`," declare any needed fields, and fill in the body of the constructor.

(c) (15 points) Also in the following class, fill in the body of the method "`doMerge`". There is space on the following page for any private helping methods you need to avoid code duplication.

```
package cs362exams;
import java.io.*;

/** A tool to merge lines of two sorted files into a third file.
 */
public class Merge {

    // DECLARE ANY FIELDS YOU MAY NEED BELOW, WITH JAVADOC COMMENTS









    /** Initialize this Merge with the given input file names
     * and output file name.
     * @param inFileName1 the name of the first input file
     * @param inFileName2 the name of the second input file
     * @param outFileName the name of the output file
     */
    /*@ requires inFileName1 != null && inFileName2 != null
      @       && outFileName != null;
      @ ensures (* this Merge object is initialized *);
      @*/
    public Merge(String inFileName1, String inFileName2, String outFileName) {









    }
```

```
/** Merge all lines of the two input files in ascending order
 * and write the merged output to the output file.
 * The output file is created if it doesn't exist already.
 * @throws IOException when some input/output problem occurs
 *
 * Note: you may use the method
 *    int compareTo(String s)
 * from java.lang.String to test whether a line is less than
 * the argument string, s, passed to the method.
 * Note: All files opened in this method must be closed by it,
 * even when an exception is thrown.
 */
/*@ ensures (* the sorted lines of the two input files are merged,
  @            in ascending order, into the newly-created output file *);
  @ signals (IOException) (* some I/O error occurred *);
  @*/
public void doMerge() throws IOException {









}
```

```
    // PUT ANY PRIVATE HELPING METHODS YOU NEED, IF ANY, BELOW



}
```

3. (50 points)

This is a problem about writing a correct implemetation of an ADT. You are to fill in the body of the methods in the class below, and also declare its fields. Be sure to declare these fields with the appropriate privacy. Following the class is a JUnit test class for it. You can refer to this JUnit test class as a supplement to the comments that specify the class.

Note: the JML "`for_example` clause used in the specification of some methods starts a section of the specification with several examples. You can also look at the JUnit tests for examples. In JML `==>` means "implies that" and `<==>` means "if and only if".

```
package cs362exams;

/** This class is responsible for tracking ranges of opinions. It might
 * be used in a statistical or polling program.  Ranges are closed
 * intervals of ints.  In English descriptions, we write [3, 7]
 * for the range that is the set {3, 4, 5, 6, 7} of measurements.
 * That is, the endpoints are included.
 */
public class Range implements Cloneable {

    // DECLARE ANY FIELDS YOU MAY NEED BELOW, WITH JAVADOC COMMENTS









    /** Initialize this Range object with the given
     * lower and upper bounds.
     * @param lower, the desired lower bound
     * @param upper, the desired upper bound
     */
    //@ requires lower <= upper;
    //@ ensures getLowerBound() == lower;
    //@ ensures getUpperBound() == upper;
    public Range(int lower, int upper) {









    }
```

```
/** Return this Range's lower bound.
 *  For example, if this is [3,7], then return 3.
 * @return the lower bound of this Range object. */
//@ ensures (* \result is the lower bound of this range *);
/*@ for_example
  @   requires this.equals(new Range(3, 7));
  @   ensures \result == 3; @*/
public /*@ pure @*/ int getLowerBound() {



}

/** Return this Range's upper bound.
 *  For example, if this is [3,7], then return 7.
 * @return the upper bound of this Range object. */
//@ ensures (* \result is the upper bound of this range *);
/*@ for_example
  @   requires this.equals(new Range(3, 7));
  @   ensures \result == 7; @*/
public /*@ pure @*/ int getUpperBound() {



}

/* Return a string containing "[", followed by the lower bound,
 * a comma and space (", "), the upper bound, and then "]".
 * For example, if this is [3,7], the result is "[3, 7]".
 * @see java.lang.Object#toString()
 */
/*@ also
  @   ensures \result != null
  @        && \result.equals("[" + getLowerBound() + ", "
  @                          + getUpperBound() + "]");
  @ for_example
  @     requires getLowerBound() == 3
  @           && getUpperBound() == 7;
  @     ensures "[3, 7]".equals(\result);
  @*/
public String toString() {




}
```

```
/** Compare this Range to the given one, and return true just when all the
 * measurements in this range are strictly less than those in the argument
 * range. For example, if this is [3,7] and r is [8,9], the return true.
 * However, if this is [3,7] and r is [6,10] or [7,9], return false.
 * @param r, the argument range (the other range).
 * @return true, if the upper bound of this is strictly less than
 *               the lower bound of r, false otherwise.
 */
/*@ public normal_behavior  // the general case
  @     requires r != null;
  @     ensures \result <==> (this.getUpperBound() < r.getLowerBound());
  @ for_example             // examples follow
  @   public normal_example
  @     requires this.getLowerBound() == 3 && this.getUpperBound() == 7
  @           && r.getLowerBound() == 8 && r.getUpperBound() == 9;
  @     ensures \result <==> true;
  @ also
  @   public normal_example
  @     requires this.getLowerBound() == 3 && this.getUpperBound() == 7
  @           && r.getLowerBound() == 6 && r.getUpperBound() == 10;
  @     ensures \result <==> false;
  @ also
  @   public normal_example
  @     requires this.getLowerBound() == 3 && this.getUpperBound() == 7
  @           && r.getLowerBound() == 7 && r.getUpperBound() == 9;
  @     ensures \result <==> false;
  @ also
  @   public normal_example
  @     requires this.getLowerBound() == 3 && this.getUpperBound() == 7
  @           && r.getLowerBound() == 1 && r.getUpperBound() == 11;
  @     ensures \result <==> false;
  @ also
  @   public normal_example
  @     requires this.getLowerBound() == 4 && this.getUpperBound() == 15
  @           && r.getLowerBound() == 0 && r.getUpperBound() == 5;
  @     ensures \result <==> false;
  @*/
public /*@ pure @*/ boolean strictlyLessThan(Range r) {




}
```

```
/** Return true just when o is a Range
 * that is not null and has the same
 * lower and upper bounds as this Range.
 * @param o the object to compare against
 * @see Object#equals(Object)
 */
//@ also
/*@   requires !(o instanceof Range);
  @   ensures \result == false;
  @ also
  @   requires o instanceof Range;
  @   ensures \result
  @      <==> (getLowerBound() == ((Range)o).getLowerBound()
  @            && getUpperBound() == ((Range)o).getUpperBound());
  @*/
public boolean equals(Object o) {









}

/** Return a hash code for this object.
 * @see java.lang.Object#hashCode()
 */
public int hashCode() {




}
```

```
    /** Return a new copy of this object,
     * that shares the same lower and upper bounds.
     * @see java.lang.Object#clone()
     */
    /*@ also
      @   ensures \result != this && \result != null;
      @   ensures ((Range)\result).getLowerBound() == getLowerBound();
      @   ensures ((Range)\result).getUpperBound() == getUpperBound();
      @*/
    public Object clone() {









   }
}
```

The following is a JUnit test class for the class above. You don't have to read this if you understand what to do already.

```
package cs362exams;

import junit.framework.TestCase;

/** Tests for the Range class */
public class RangeTest extends TestCase {

    /**Constructor for RangeTest. */
    public RangeTest(String name) {
        super(name);
    }

    /** Run the tests. */
    public static void main(String[] args) {
        junit.textui.TestRunner.run(RangeTest.class);
    }

    /** Lower and upper bounds for use in testing. */
    private int[][] bounds = {
        {-1, 1},
        {0, 0},
        {2, 7},
        {2, 8},
        {3, 6},
        {3, 8},
        {3, 7},  // our favorite example :-)
        {4, 6},
        {4, 7},
        {5000, 700000},
    };

    /** Range objects for use in testing.
     * These are initialized by the initializer block
     * that follows this declaration.
     */
    private Range[] receivers;

    {   // initializer block that initializes receivers
        receivers = new Range[bounds.length];
        for (int i = 0; i < bounds.length; i++) {
            receivers[i] = new Range(bounds[i][0],
                                     bounds[i][1]);
        }
    }
```

```
//@ private invariant bounds.length == receivers.length;

/** Test the Range constructor */
public void testRange() {
   Range r = new Range(3, 7);
    assertTrue(r.getLowerBound() == 3);
    assertTrue(r.getUpperBound() == 7);
}

/** Test the hashCode method */
public void testHashCode() {
    for (int i = 0; i < receivers.length; i++) {
        assertEquals(receivers[i].hashCode(),
                     receivers[i].hashCode());
        assertEquals(receivers[i].hashCode(),
                     receivers[i].clone().hashCode());
    }
}

/** Test the getLowerBound method */
public void testGetLowerBound() {
    for (int i = 0; i < receivers.length; i++) {
        assertEquals(bounds[i][0],
                     receivers[i].getLowerBound());
    }
}

/** Test the getUpperBound method */
public void testGetUpperBound() {
    for (int i = 0; i < receivers.length; i++) {
        assertEquals(bounds[i][1],
                     receivers[i].getUpperBound());
    }
}

/** Test for String toString() */
public void testToString() {
    for (int i = 0; i < receivers.length; i++) {
        assertEquals("[" + bounds[i][0]
                     + ", " + bounds[i][1] + "]",
                     receivers[i].toString());
    }
}
```

```
    /** Test for strictlyLessThan. */
    public void testStrictlyLessThan() {
        Object n = null;
        for (int i = 0; i < receivers.length; i++) {
            for (int j = 0; j < receivers.length; j++) {
                assertTrue(receivers[i].strictlyLessThan(receivers[j])
                           == (receivers[i].getUpperBound()
                               < receivers[j].getLowerBound()));
            }
        }
    }


    /** Test for equals. */
    public void testEquals() {
        for (int i = 0; i < receivers.length; i++) {
            assertFalse(receivers[i].equals(null));
            assertFalse(receivers[i].equals(new Integer(362)));
            for (int j = 0; j < receivers.length; j++) {
                // all the receivers have some difference in their bounds
                assertEquals(i == j,
                             receivers[i].equals(receivers[j]));
            }
        }
    }

    /** Test for clone. */
    public void testClone() {
        for (int i = 0; i < receivers.length; i++) {
            assertTrue(receivers[i].clone() != receivers[i]);
            assertTrue(receivers[i].clone().equals(receivers[i]));
        }
    }
}
```