Com S 362
Fall 2003

Name: _____

Object-Oriented Analysis and Design

Exam 1 on Abstract Data Types and Java

This test has 3 questions and pages numbered 1 through 14.

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) Briefly describe a good reason for designing a method that throws a checked exception in Java. Given an example.

2. This is a problem about simple applications, file I/O, and exceptions in Java. In this problem you will write your code by filling in the empty methods in the code below. The program consists of 2 classes, found below. The details are in the program's comments. (An overview can be found in part (c) below, which contains the main class and the main method.)

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

   (b) (15 points) Also in the following class, fill in the body of the method "`doIt`".

```
package cs362exams;

import java.io.*;

/** A tool to produce a slice of a file,
 * consisting of lines in the file that follow
 * a line with a given pattern.
 */
public class FileAfter {

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









    /** Initialize this FileAfter with the given input file name,
     * pattern, and output file name.
     * @param inFileName the name of the input file.
     * @param pattern a string sought at the beginning of a line
     * @param outFileName the name of the output file
     */
    /*@ requires inFileName != null && pattern != null
      @       && outFileName != null;
      @*/
    public FileAfter(String inFileName, String pattern, String outFileName) {









    }
```

```
    /** Copying all lines of the input file
     * following the first line (if any) in the input file
     * that starts with the pattern to the output file.
     * These lines, that is the ones following the first line
     * matching the pattern, are put into the output file.
     * @throws IOException when some input/output problem occurs
     *
     * Note: you may use the method
     *    boolean startsWith(String prefix)
     * from java.lang.String to test whether a line starts with
     * the argument string, prefix, passed to the method.
     * Note: All files opened in this method must be closed by it,
     * even when an exception is thrown.
     */
    public void doIt() throws IOException {









    }


}
```

(c) (10 points) In the following class, fill in the body of the method "`main`". (There is more space on the next page.)

```
package cs362exams;

import java.io.IOException;

/** Copy the contents of the file named in the first argument
 * starting at the first line following the line matching the string
 * given in the second argument (if any) to the file named
 * in the third argument. If there is any I/O error,
 * then an error message is printed to System.err.
 * and the application exits with a non-zero exit code. */
public class FileAfterMain {

    /** Run this application.  This method is responsible for calling
     * the doIt() method of the class FileAfter. This method is also
     * responsible for catching any IOExceptions that doIt() may throw;
     * if it catches any such exceptions, then it should print
     * their message to System.err, preceeded by
     *    "FileAfterMain: I/O error: "
     * and followed by a newline.  This method must also call the
     * checkArguments method and exit with a non-zero exit code if
     * there are not three arguments in argv.
     * @param argv contains the input file name,
     *        the string to match, and the output file name. */
    //@ requires argv != null && \nonnullelements(argv);
    public static void main(String[] argv) {









    }
```

```
    /** Check for proper usage of arguments,
     * and issue appropriate error messages.
     * @return false just when there are errors in the arguments. */
    //@ ensures !\result || (argv.length == 3);
    private static boolean checkArguments(String[] argv) {
        if (argv.length != 3) {
            System.err.println("FileAfterMain: expecting 3 arguments");
            return false;
        } else {
            return true;
        }
    }

}
```

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 information about
 * a book.  It might be used in a library catalog system.
 */
public class Book implements Cloneable, Comparable {

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




    /** Initialize this Book object with the given
     * author name and title.
     * @param authorName the author's name
     * @param title the book's title
     */
    //@ requires authorName != null && title != null;
    //@ ensures getAuthorName().equals(authorName);
    //@ ensures getTitle().equals(title);
    public Book(String authorName, String title) {



    }

    /** Return this book's author's name. */
    //@ ensures \result != null;
    public /*@ pure @*/ String getAuthorName() {


    }
```

```
/** Return this book's title. */
//@ ensures \result != null;
public /*@ pure @*/ String getTitle() {


}

/* Return a string containing the title,
 * the string ", by ",
 * and then the book's author's name.
 * For example, one famous book would produce the string:
 *    "Wuthering Heights, by Emily Bronte"
 * @see java.lang.Object#toString() */
/*@ also
  @   ensures \result != null
  @        && \result.equals(getTitle() + ", by "
  @                          + getAuthorName());
  @ for_example
  @   public normal_example
  @     requires getAuthorName().equals("Emily Bronte")
  @           && getTitle().equals("Wuthering Heights");
  @     ensures "Wuthering Heights, by Emily Bronte"
  @             .equals(\result);
  @*/
public String toString() {



}

/** Compare this book to the given object, using
 * first the author's name, and then the title.
 * @see Comparable#compareTo(Object) */
/*@ also
  @   requires o instanceof Book;
  @   ensures \result == compareTo((Book)o);
  @ also
  @   requires !(o instanceof Book);
  @   signals (ClassCastException e) e != null;
  @*/
public /*@ pure @*/ int compareTo(Object o) {




}
```

```
/** Compare this book to the given book,
 * using the first the author's name, then the title.
 * @param b the book to compare against
 * @return an integer, with a negative result meaning
 *         this book compares strictly less than b,
 *         0 meaning they are equal, and
 *         a positive result meaning this book is strictly
 *         greater than b.
 * Note: you may want to use String's compareTo method.
 */
/*@   requires b != null;
  @   ensures \result < 0
  @       ==> (getAuthorName().compareTo(b.getAuthorName()) < 0
  @            || (getAuthorName().equals(b.getAuthorName())
  @                && getTitle().compareTo(b.getTitle()) < 0));
  @   ensures \result == 0 ==> this.equals(b);
  @   ensures \result > 0
  @       ==> (getAuthorName().compareTo(b.getAuthorName()) > 0
  @            || (getAuthorName().equals(b.getAuthorName())
  @                && getTitle().compareTo(b.getTitle()) > 0));
  @ for_example
  @   public normal_example
  @     requires getAuthorName().equals("Alexandre Dumas")
  @           && b.getAuthorName().equals("Mark Twain");
  @     ensures \result < 0;
  @ also
  @   public normal_example
  @     requires getAuthorName().equals("Jane Austin")
  @           && b.getAuthorName().equals("Jane Austin")
  @           && getTitle().equals("Emma")
  @           && b.getTitle().equals("Mansfield Park");
  @     ensures \result < 0;
  @ also
  @   public normal_example
  @     requires getAuthorName().equals("Jane Austin")
  @           && b.getAuthorName().equals("Jane Austin")
  @           && getTitle().equals("Emma")
  @           && b.getTitle().equals("Emma");
  @     ensures \result == 0;
  @ also
  @   public normal_example
  @     requires getAuthorName().equals("Jane Austin")
  @           && b.getAuthorName().equals("Jane Austin")
  @           && getTitle().equals("Sense and Sensibility")
  @           && b.getTitle().equals("Emma");
  @     ensures \result > 0;
  @ also
```

```
  @   public normal_example
  @     requires getAuthorName().equals("Mark Twain")
  @           && b.getAuthorName().equals("Jane Austin");
  @     ensures \result > 0;
  @*/
public /*@ pure @*/ int compareTo(Book b) {









}

/** Return true just when o is a Book
 * that is not null and has the same
 * author and title as this book.
 * @param o the object to compare against
 * @see Object#equals(Object)
 */
//@ also
/*@   requires !(o instanceof Book);
  @   ensures \result == false;
  @ also
  @   requires o instanceof Book;
  @   ensures \result
  @      <==> (getAuthorName()
  @                  .equals(((Book)o).getAuthorName())
  @            && getTitle().equals(((Book)o).getTitle()));
  @*/
public boolean equals(Object o) {







}
```

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




    }

    /** Return a new copy of this object,
     * that shares the same author and title.
     * @see java.lang.Object#clone()
     */
    /*@ also
      @   ensures \result != this && \result != null;
      @   ensures ((Book)\result).getAuthorName()
      @           .equals(getAuthorName());
      @   ensures ((Book)\result).getTitle().equals(getTitle());
      @*/
    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 Book class */
public class BookTest extends TestCase {

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

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

    /** Authors and titles for use in testing. */
    private String[][] authorTitle = {
        {"Harry Turtledove", "In the Balance"},
        {"Harry Turtledove", "Tilting the Balance"},
        {"Harry Turtledove", "Upsetting the Balance"},
        {"Fyodor Dostoyevsky", "The Brothers Karamazov"},
        {"Agatha Christie", "And Then There Were None"},
        {"Agatha Christie", "Murder on the Orient Express"},
    };

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

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

    //@ private invariant authorTitle.length == receivers.length;

    /** Test the Book constructor */
    public void testBook() {
```

```
    String a = "Jane Austen";
    String t = "Northangar Abbey";
    Book b = new Book(a, t);
    assertTrue(b.getAuthorName().equals(a));
    assertTrue(b.getTitle().equals(t));
}

/** 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 getAuthorName method */
public void testGetAuthorName() {
    for (int i = 0; i < receivers.length; i++) {
        assertEquals(authorTitle[i][0],
                     receivers[i].getAuthorName());
    }
}

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

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

/** Test for int compareTo(Object). */
public void testCompareToObject() {
    Object n = null;
    for (int i = 0; i < receivers.length; i++) {
        try {
            receivers[i].compareTo(n);
```

```
            fail("compareTo did not throw a ClassCastException");
        } catch (ClassCastException e) {
            // ok
        }
        try {
            receivers[i].compareTo(new Integer(362));
            fail("compareTo did not throw a ClassCastException");
        } catch (ClassCastException e) {
            // ok
        }

        for (int j = 0; j < receivers.length; j++) {
            assertTrue(receivers[i].compareTo((Object)receivers[j])
                       == receivers[i].compareTo(receivers[j]));
        }
    }
}

/** Test for int compareTo(Book). */
public void testCompareToBook() {
    for (int i = 0; i < receivers.length; i++) {
        for (int j = 0; j < receivers.length; j++) {
            int res = receivers[i].compareTo(receivers[j]);
            if (res == 0) {
                assertEquals(receivers[i].getAuthorName(),
                             receivers[j].getAuthorName());
                assertEquals(receivers[i].getTitle(),
                             receivers[j].getTitle());
            }
            if (res < 0) {
                assertTrue(receivers[i].getAuthorName()
                                .compareTo(receivers[j].getAuthorName()) < 0
                           || (receivers[i].getAuthorName()
                                 .equals(receivers[j].getAuthorName())
                               && receivers[i].getTitle()
                                  .compareTo(receivers[j].getTitle())
                                   < 0));
            }
            if (res > 0) {
                assertTrue(receivers[i].getAuthorName()
                                .compareTo(receivers[j].getAuthorName()) > 0
                           || (receivers[i].getAuthorName()
                                 .equals(receivers[j].getAuthorName())
                               && receivers[i].getTitle()
                                  .compareTo(receivers[j].getTitle())
                                   > 0));
            }
        }
```

```
        }
    }

    /** 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++) {
                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]));
        }
    }
}
```