Com S 362 Name: ____________________
Fall 2002

Object-Oriented Analysis and Design

Practice for Exam 1 on Abstract Data Types and Java

This test has 3 questions and pages numbered 1 through 17.

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 answer the following questions:

   (a) Why is it a bad idea to use public (non-final) fields in a Java class?

   (b) For what kinds of programming projects would OO techniques not be useful? Give an example.

2. (40 points) 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 (b) below, which contains the main class and the main method.)

   (a) (5 points) In the following class, "`FileCatter`," fill in the body of the constructor.

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

```
package cs362practice;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

/** Objects that can cat a given file's contents to
 * standard output.
 * @author Gary T. Leavens
 */
public class FileCatter {

    /** The name of the file to cat.
     * We store just the name so that we don't get errors
     * unless and until the file is catted, and so no resources
     * are consumed until then. */
    private String name;

    /**
     * Initialize this FileCatter with the given name
     * @param name the file name
     */
    //@ requires name != null;
    public FileCatter(String name) {


    }

    /**
     * Open the file with the given name,
     * read each line of the file and write it to System.out,
     * and then close the file.  The file should be closed in
     * all cases.   You can use the expression
     *    new BufferedReader(new FileReader(inFile))
     * to create a BufferedReader on the file inFile.
     * We expect that you know how to create a File object
     * from a String so that it is open on the file named by the String.
     * Be sure to close the file when you are done.
     * @throws IOException if any I/O problems arise
     */
    public void catToStdout() throws IOException {















    }
}
```

(c) (20 points) In the following class, fill in the body of the method "`main`".

```
package cs362practice;

import java.io.IOException;

/** Concatenate the text files named on the command line
 * to standard output.
 * @author Gary T. Leavens
 */
public class Cat {

    /** Run this application.  This method is responsible for calling
     *  the catToStdout method of the class FileCatter, which should be used
     *  to copy the contents of each named argument to standard output.
     *  This method is also responsible for catching any IOExceptions
     *  that the catToStdout method may throw,
     *  and if it catches them it should print their message to System.err,
     *  preceeded by "Cat: I/O error: " and followed by a newline.
     *  IOExceptions should not cause the program to exit immediately;
     *  instead, each error should be noted, and the program
     *  should continue with the next file (if any).  When the program
     *  has worked on all the files, it should exit with a non-zero
     *  status code if any IOExceptions were encountered.
     * @param args the names of the text files to copy to System.out
     */
    //@ requires args != null && args.length > 0;
    //@ requires \nonnullelements(args);
    public static void main(String[] args) {













    }
}
```

3. This is a problem about writing a correct implemetation of an ADT. You are to fill in the implementations of two classes, `Rectangular` and `Polar` that implement the interface `Complex` found below. The interface `Complex` contains the specification of what you are to do; read it and then continue with the implementation parts of this problem that follow it. Following the classes is a JUnit test class. You can refer to this JUnit test class as a supplement to the comments that specify the class.

```
package cs362practice;

//@ model import org.jmlspecs.models.JMLDouble;

/** Complex numbers.  Note that these are immutable.
 * Abstractly, one can think of a complex number as
 *  realPart+(imaginaryPart*i).
 * Alternatively, one can think of it as distance
 * from the origin along a given angle
 * (measured in radians counterclockwise from
 * the positive x axis), hence a pair of
 *  (magnitude, angle).
 * This class supports both of these views.
 */
public /*@ pure @*/ interface Complex {

    //@ public ghost static final double tolerance = 1e-10;

    /** Return the real part of this complex number. */
    /*@ ensures JMLDouble.withinEpsilonOf(
      @             magnitude()*Math.cos(angle()),
      @             \result,
      @             tolerance);
      @*/
    double realPart();

    /** Return the imaginary part of this complex number. */
    /*@ ensures JMLDouble.withinEpsilonOf(
      @             \result,
      @             magnitude()*Math.sin(angle()),
      @             tolerance);
      @*/
    double imaginaryPart();

    /** Return the magnitude of this complex number. */
    /*@ ensures JMLDouble.withinEpsilonOf(
      @             Math.sqrt(realPart()*realPart()
      @                       + imaginaryPart()*imaginaryPart()),
      @             \result,
      @             tolerance);
      @*/
```

```
double magnitude();

/** Return the magnitude of this complex number. */
/*@ ensures JMLDouble.withinEpsilonOf(
  @             Math.atan2(imaginaryPart(), realPart()),
  @             \result,
  @             tolerance);
  @*/
double angle();

/** Return this + b (the sum of this and b). */
//@ requires b != null;
//@ ensures \result != null;
/*@ ensures JMLDouble.withinEpsilonOf(
  @             this.realPart() + b.realPart(),
  @             \result.realPart(),
  @             tolerance);
  @ ensures JMLDouble.withinEpsilonOf(
  @             this.imaginaryPart() + b.imaginaryPart(),
  @             \result.imaginaryPart(),
  @             tolerance);
  @*/
Complex add(Complex b);

/** Return this - b (the difference between this and b). */
//@ requires b != null;
//@ ensures \result != null;
/*@ ensures JMLDouble.withinEpsilonOf(
  @             this.realPart() - b.realPart(),
  @             \result.realPart(),
  @             tolerance);
  @ ensures JMLDouble.withinEpsilonOf(
  @             this.imaginaryPart() - b.imaginaryPart(),
  @             \result.imaginaryPart(),
  @             tolerance);
  @*/
Complex sub(Complex b);

/** Tell whether the given angles are the same, taking into account
 *  that angles measured in radians wrap around after 2*Math.PI times.
 */
/*@ public model pure boolean similarAngle(double ang1, double ang2) {
  @    ang1 = positiveRemainder(ang1, 2*Math.PI);
  @    ang2 = positiveRemainder(ang2, 2*Math.PI);
  @    return JMLDouble.withinEpsilonOf(ang1, ang2, tolerance);
  @ }
  @*/
```

```
    /** Return the positive remainder of n divided by d. */
    /*@ ensures \result >= 0.0;
      @ public model pure double positiveRemainder(double n, double d) {
      @    n = n % d;
      @    if (n < 0) {
      @       n += d;
      @    }
      @    return d;
      @ }
      @*/

    /** Return this * b (the product of this and b). */
    //@ requires b != null;
    //@ ensures \result != null;
    /*@ ensures JMLDouble.withinEpsilonOf(
      @             this.magnitude() * b.magnitude(),
      @             \result.magnitude(),
      @             tolerance);
      @ ensures similarAngle(this.angle() + b.angle(),
      @                      \result.angle());
      @*/
    Complex mul(Complex b);

    /** Return this/b (the quotient of this by b). */
    //@ requires b != null && b.magnitude() != 0;
    //@ ensures \result != null;
    /*@ ensures JMLDouble.withinEpsilonOf(
      @             this.magnitude() / b.magnitude(),
      @             \result.magnitude(),
      @             tolerance);
      @ ensures similarAngle(this.angle() - b.angle(),
      @                      \result.angle());
      @*/
    Complex div(Complex b);

    /** Return true if these are the same complex number. */
    /*@ also
      @ ensures \result
      @     <==> o instanceof Complex
      @          && this.realPart() == ((Complex)o).realPart()
      @          && this.imaginaryPart() == ((Complex)o).imaginaryPart();
      @ ensures \result
      @     <==> o instanceof Complex
      @          && this.magnitude() == ((Complex)o).magnitude()
      @          && this.angle() == ((Complex)o).angle();
      @*/
    boolean equals(Object o);
}
```

(a) (25 points) In the following class, declare any necessary fields, and then fill in the bodies of all the constructors and methods that have no code.

```
package cs362practice;

/** Complex numbers in rectangular coordinates.
 */
public /*@ pure @*/ class Rectangular implements Complex {
    // DECLARE ANY FIELDS YOU MAY NEED BELOW





    /** Initialize this Complex number to be 0+(0*i). */
    //@ ensures realPart() == 0.0 && imaginaryPart() == 0.0;
    public Rectangular() {


    }

    /** Initialize this Complex number to be re+(0*i). */
    //@ ensures realPart() == re && imaginaryPart() == 0.0;
    public Rectangular(double re) {


    }

    /** Initialize this Complex number to be re+(img*i). */
    //@ ensures realPart() == re && imaginaryPart() == img;
    public Rectangular(double re, double img) {


    }

    /** @see cs362practice.Complex#realPart() */
    public double realPart() {


    }

    /** @see cs362practice.Complex#imaginaryPart() */
    public double imaginaryPart() {


    }

    /** @see cs362practice.Complex#magnitude() */
```

```
public double magnitude() {


}

/** @see cs362practice.Complex#angle() */
public double angle() {


}

/** @see cs362practice.Complex#add(Complex) */
public Complex add(Complex b) {




}

/** @see cs362practice.Complex#sub(Complex) */
public Complex sub(Complex b) {




}

/** @see cs362practice.Complex#mul(Complex) */
public Complex mul(Complex b) {




}

/** @see cs362practice.Complex#div(Complex) */
public Complex div(Complex b) {




}

/** @see cs362practice.Complex#equals(Object) */
public boolean equals(Object o) {
    }

    public String toString() {
        return realPart() + " + " + imaginaryPart() + "*i";
    }
}
```

(b) (25 points) In the following class, declare any necessary fields, and then fill in the bodies of all the constructors and methods that have no code.

```
package cs362practice;

/**
 * @author Gary T. Leavens
 */
public /*@ pure @*/ class Polar implements Complex {

    // DECLARE ANY FIELDS YOU MAY NEED BELOW







    /** Initialize this polar coordinate number
     * with magnitude mag and angle ang, except that
     * when the magnitude is negative, this
     * is interpreted as magnitude -mag and angle ang+Math.PI.
     * @param mag the magnitude desired
     * @param ang the angle in radians, measured
     *            counterclockwise from the positive x axis
     */
    /*@   requires mag >= 0;
      @   ensures this.magnitude() == mag;
      @   ensures this.angle() == standardizeAngle(ang);
      @ also
      @   requires mag < 0;
      @   ensures this.magnitude() == - mag;
      @   ensures this.angle() == standardizeAngle(ang+Math.PI);
      @*/
    public Polar(double mag, double ang) {







    }

    /** Standardize the angle so it’s between
     * -Math.PI and Math.PI (radians).
     */
    public static /*@ pure @*/ double standardizeAngle(double rad) {
        rad = rad % (2*Math.PI);
        if (rad > Math.PI) {
```

```
        return rad - 2*Math.PI;
    } else if (rad < -Math.PI) {
        return rad + 2*Math.PI;
    } else {
        return rad;
    }
}

/** @see cs362practice.Complex#realPart() */
public double realPart() {


}

/** @see cs362practice.Complex#imaginaryPart() */
public double imaginaryPart() {


}

/** @see cs362practice.Complex#magnitude() */
public double magnitude() {


}

/** @see cs362practice.Complex#angle() */
public double angle() {


}

/** @see cs362practice.Complex#add(Complex) */
public Complex add(Complex b) {




}

/** @see cs362practice.Complex#sub(Complex) */
public Complex sub(Complex b) {




}
```

```
    /** @see cs362practice.Complex#mul(Complex) */
    public Complex mul(Complex b) {




    }

    /** @see cs362practice.Complex#div(Complex) */
    public Complex div(Complex b) {




    }

    /** @see cs362practice.Complex#equals(Object) */
    public boolean equals(Object o) {








    }

    public String toString() {
        return "(" + magnitude() + ", " + angle() + ")";
    }
}
```

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

```
package cs362practice;

import junit.framework.TestCase;

/** Test for the complex number types. */
public class ComplexTest extends TestCase {

    /** Constructor for ComplexTest.
     * @param name
     */
    public ComplexTest(String name) {
        super(name);
    }

    public static void main(String[] args) {
        junit.textui.TestRunner.run(ComplexTest.class);
    }

    private static final double tolerance = 1e-10;
    private Complex[] receivers;

    /** Initalize the receivers array. */
    protected void setUp() throws Exception {
        if (receivers == null) {
            receivers =
                new Complex[] {
                    new Rectangular(0.0, 0.0),
                    new Rectangular(2.0, 0.0),
                    new Rectangular(2.1, 3.4),
                    new Rectangular(-10.0, 75.2),
                    new Rectangular(-57.34, -0.1e-10),
                    new Polar(0.0, 0.0),
                    new Polar(2.1, Math.PI / 3),
                    new Polar(7.5, Math.PI - 0.02),
                    new Polar(-10.6, 3 * Math.PI / 2),
                    };
        }
    }

    /** Test the method Polar.standardizeAngle. */
    public void testStandardizeAngle() {
        assertEquals(0.0, Polar.standardizeAngle(0.0), tolerance);
        assertEquals(3.0, Polar.standardizeAngle(3.0), tolerance);
        assertEquals(-0.5*Math.PI, Polar.standardizeAngle(1.5*Math.PI),
                     tolerance);
        assertEquals(0.5*Math.PI, Polar.standardizeAngle(-1.5*Math.PI),
                     tolerance);
        assertEquals(0.5*Math.PI, Polar.standardizeAngle(6.5*Math.PI),
                     tolerance);
        assertEquals(0.5*Math.PI, Polar.standardizeAngle(-7.5*Math.PI),
                     tolerance);
    }

    /** Test the method realPart. */
    public void testRealPart() {
        for (int i = 0; i < receivers.length; i++) {
            assertEquals(
                receivers[i].magnitude() * Math.cos(receivers[i].angle()),
                receivers[i].realPart(),
                tolerance);
        }
    }

    /** Test the method imaginaryPart. */
    public void testImaginaryPart() {
        for (int i = 0; i < receivers.length; i++) {
            assertEquals(
                receivers[i].magnitude() * Math.sin(receivers[i].angle()),
                receivers[i].imaginaryPart(),
                tolerance);
        }
    }

    /** Test the method magnitude. */
    public void testMagnitude() {
        for (int i = 0; i < receivers.length; i++) {
            assertEquals(
                Math.sqrt(
                    receivers[i].realPart() * receivers[i].realPart()
                        + receivers[i].imaginaryPart()
                            * receivers[i].imaginaryPart()),
                receivers[i].magnitude(),
                tolerance);
        }
    }

    /** Test the method angle. */
    public void testAngle() {
        for (int i = 0; i < receivers.length; i++) {
            assertEquals(
                "Angle not right for " + receivers[i],
                Math.atan2(
                    receivers[i].imaginaryPart(),
                receivers[i].realPart()
                ),
            receivers[i].angle(),
            tolerance);
    }
}

/** Test the method add. */
public void testAdd() {
    for (int i = 0; i < receivers.length; i++) {
        for (int j = 0; j < receivers.length; j++) {
            assertEquals(
                "Add real part not right for " + receivers[i]
                + " and " + receivers[j],
                receivers[i].realPart() + receivers[j].realPart(),
                receivers[i].add(receivers[j]).realPart(),
                tolerance);
            assertEquals(
                receivers[i].imaginaryPart() + receivers[j].imaginaryPart(),
                receivers[i].add(receivers[j]).imaginaryPart(),
                tolerance);
        }
    }
}

/** Test the method sub. */
public void testSub() {
    for (int i = 0; i < receivers.length; i++) {
        for (int j = 0; j < receivers.length; j++) {
            assertEquals(
                "Sub real part not right for " + receivers[i]
                + " and " + receivers[j],
                receivers[i].realPart() - receivers[j].realPart(),
                receivers[i].sub(receivers[j]).realPart(),
                tolerance);
            assertEquals(
                receivers[i].imaginaryPart() - receivers[j].imaginaryPart(),
                receivers[i].sub(receivers[j]).imaginaryPart(),
                tolerance);
        }
    }
}

/** Test the method mul. */
public void testMul() {
    for (int i = 0; i < receivers.length; i++) {
        for (int j = 0; j < receivers.length; j++) {
            assertEquals(
                    receivers[i].magnitude() * receivers[j].magnitude(),
                    receivers[i].mul(receivers[j]).magnitude(),
                    tolerance);
                assertEquals(
                    Polar.standardizeAngle(receivers[i].angle()
                                           + receivers[j].angle()),
                    receivers[i].mul(receivers[j]).angle(),
                    tolerance);
            }
        }
    }

    /** Test the method div. */
    public void testDiv() {
        for (int i = 0; i < receivers.length; i++) {
            for (int j = 0; j < receivers.length; j++) {
                if (receivers[j].magnitude() != 0) {
                    assertEquals(
                        receivers[i].magnitude() / receivers[j].magnitude(),
                        receivers[i].div(receivers[j]).magnitude(),
                        tolerance);
                    assertEquals(
                        Polar.standardizeAngle(receivers[i].angle()
                                               - receivers[j].angle()),
                        receivers[i].div(receivers[j]).angle(),
                        tolerance);
                }
            }
        }
    }

    /** Test the method equals. */
    public void testEquals() {
        for (int i = 0; i < receivers.length; i++) {
            assertFalse(receivers[i].equals(null));
            assertFalse(receivers[i].equals(new Double(2.0)));
            for (int j = 0; j < receivers.length; j++) {
                assertEquals(
                    i == j
                        || (receivers[i].realPart() == receivers[j].realPart()
                            && receivers[i].imaginaryPart()
                                == receivers[j].imaginaryPart()),
                    receivers[i].equals(receivers[j]));
            }
        }
    }
}
```