Name: ____________________________________

CSC 102 Midterm Exam

The exam is open-book and open-note. Questions 1 through 5 are to be answered on the exam paper itself. Question 6 is to be submitted via handin. If you have time after completing Question 6, you may check your answers to Questions 1 thorough 5 as you see fit, in the lab.

The exam is worth 150 points total. Individual point values are indicated for each question.

1. (20 points) Given below is an incomplete definition for a class named `Planet`. There are four places for you to fill in code, indicated by the comments `/* Define ... here. */`. The code you fill in must meet the following specification:

   a. The `Planet` class has two private data fields, both of type `double`. The fields hold values for the *mass* and *diameter* of the planet.

   b. `Planet` has a default constructor that initializes the value of the mass to $5.98 * 10^{24}$, and initializes the value of the diameter to 12,756.2. (These values happen to be the mass and diameter of the earth.)

   c. `Planet` has an initializing constructor that takes parameters for the mass and diameter. It initializes the data fields to the values of the parameters.

   d. `Planet` has a public method `density`. It takes no parameters and returns `double`. The method computes the density of the planet using the following formula:

      density = mass / volume

      where *volume* = $4/3 * PI * r^3$, with *r* as the radius (i.e., diameter / 2). You may use `Math.PI` in your code, without an import declaration.

   Here is the class with the code to be filled in:

```
public class Planet {

    /* Define private data fields here (2 points). */









    /* Define default constructor here (4 points). */











    /* Define initializing constructor here (6 points). */
    /* Define density method here (8 points). */









}
```

2. (16 points) Given below is an incomplete definition for a class named `PlanetTester`. It is a testing program for the `Planet` class you defined in the previous question. Fill in the body of the main method to perform the following testing:

   a. Declare and initialize a `Planet` variable named `p1`, using the default constructor.

   b. Declare and initialize a `Planet` variable named `p2`, using the initializing constructor with values `1.5e28` and `25000`.

   c. Print out the density of `Planet p1`.

   d. Print out the density of `Planet p2`.

   Note that you do NOT need to print out extra lines that say "`Expect ...`", as is done in some of the testing programs we've used. Your print statements here need only be one line each.

```
public class PlanetTester {

    public static void main(String[] args) {

    // Fill in code here, per the description above.











    }
}
```

3. Make the following upgrades to your definitions of `Planet` and `PlanetTester` in the preceding questions:
   a. (8 points) Fill in the following definition of `Planet.toString()`. The method returns a string like this:

      `mass:` *mass*`, diameter:` *diameter*

      For example, toString called on a planet created with the default constructor returns the string

      ```
      mass: 5.98E24, diameter: 12756.2
      ```

      Here is the definition to be filled in; don't forget that the method is being defined within class `Planet`.

      ```
      public String toString() {

          /* Fill in code here. */

      }
      ```

   b. (20 points) Fill in the following definition of `Planet.equals(Object)`. As indicated by the signature, the method specializes `equals` defined in `java.lang.Object`, as was done in Program 3. The definition of `Planet.equals` returns true if two planets have the same mass and diameter values. The method returns false if the `Object` parameter is null or not a `planet`.

      Here is the definition to be filled in; again, this is a member method defined within class `Planet`:

      ```
      public boolean equals(Object other) {

      }
      ```

c. (16 points) Add exactly four more test cases to your implementation of `PlanetTester.main` in Question 2. The new tests call `Planet.toString` and `Planet.equals` to test their output. The test cases should do what you consider to be the most adequate testing possible, given the constraint of testing two methods with only four test cases. (Hint: The call `p1.equals(p1)` is not an adequate test case under this constraint.)

Fill in your answer here; remember that you're adding code to the `main` method you wrote in Question 2:

4. (10 points) Given below is the code for `Circle.java`. It is part of the solution for programming assignment 3. For brevity, the comments have been left out, as has the definition of the constructor.

Fill in the following table, indicating which `Circle` methods are mutating versus which methods are non-mutating. Do so by putting an "X" in the appropriate column of the table for each of the listed methods.

method	**mutating**	**non-mutating**
getRadius		
setRadius		
getPosition		
equals		
getArea		
getColor		
setColor		
getFilled		
setFilled		
move		

Here is the code for `Circle.java`:

```
public class Circle implements Shape {

    private Color color;
    private boolean filled;
    private double radius;
    private Point position;
    public double getRadius() {
         return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

    public Point getPosition() {
        return position;
    }

    public boolean equals(Object other) {

       if ((other == null) || ! (other instanceof Circle)) {
          return false;
       }

       Circle c = (Circle) other;
       return
           color.equals(c.color) &&
           filled == c.filled &&
           position.equals(c.position) &&
           radius == c.radius;
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }

    public Color getColor() {
        return color;
    }

    public void setColor(Color color) {
        this.color = color;
    }

    public boolean getFilled() {
        return filled;
    }

    public void setFilled(boolean filled) {
        this.filled = filled;
    }

    public void move(Point delta) {
        position.x += delta.x;
        position.y += delta.y;
    }

}
```

5. (10 points) Given below are some interface and class definitions. Based on these definitions, fill in the second column of the following table, for each line of code in the `TestInterfaces.main` method. Use one of the following three indicators:
   - OK = the line compiles and runs
   - CE = the compiler gives an error for the line
   - RE = the line compiles, but produces a run-time error, i.e., throws an exception

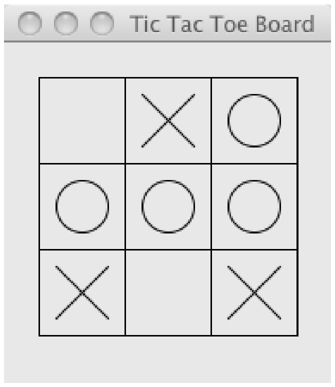
Line from main	**OK or CE or RE**
`movable.move(10);`	
`System.out.println(alive.age());`	
`System.out.println(dog1.age());`	
`car.move(10);`	
`dog1.move(car.location);`	
`alive = new Tree();`	
`dog1.sniff(dog2);`	
`dog1.sniff(dog1);`	
`dog1.sniff(car);`	
`dog2.sniff(tree);`	

Here is the code for the interfaces and classes, with some comments omitted for brevity:

```
public interface Movable {
    /** Move the given distance. */
    void move(int distance);
}

public interface Alive {
    /** Return the age in years. */
    int age();
}

public class Car implements Movable {
    /** Location reprsenting miles from the north pole. */
    private int location = 0;

    public void move(int distance) {
       location += distance;
    }
}

public class Dog implements Movable, Alive {
    /** Location represening miles from home. */
    private int location;

    /** Age in dog years. */
    private int age;

    public Dog(int location, int age) {
        this.location = location;
        this.age = age;
    }
    public void move(int distance) {
        location += distance;
    }

    public int age() {
        return this.age;
    }

    public void sniff(Alive other) {
        Dog fellowDog = (Dog) other;
        if (age >= other.age()) {
            System.out.println("Sniff.");
        }
        else {
            System.out.println("Sniff, sniff.");
        }
    }
}

public class Tree implements Alive {
    /** Age in years */
    private int age = 0;

    public int age() {
        return this.age;
    }
}

public class TestInterfaces {

    public static void main(String[] args) {
       Movable movable;
       Alive alive = new Tree();
       Car car = new Car();
       Dog dog1 = new Dog(0,5);
       Dog dog2 = new Dog(5,10);
       Tree tree = new Tree();

       movable.move(10);
       System.out.println(alive.age());
       System.out.println(dog1.age());
       car.move(10);
       dog1.move(car.location);
       alive = new Tree();
       dog1.sniff(dog2);
       dog1.sniff(dog1);
       dog1.sniff(car);
       dog2.sniff(tree);
    }
}
```

Programming Portion of the Midterm, to Be Completed in Lab

6. (50 points) Implement a program that draws a tic-tac-toe board in the following configuration:



The program must be defined in two `.java` files named `TicTacToe.java` and `TicTacToeComponent.java`. The class `TicTacToe.java` must have the `main` method. This two-class design is like the book examples that most of you used as the starting point for Program 2, and Quiz 2.

Specification details are these:

- The width (and therefore height) of each square is 50 pixels.
- The left and top margins are 20 pixels each.
- The bottom and right margins are at least 20 pixels, but may be more.
- The width and height of the framing rectangle for both the X's and O's is 30 pixels.
- Each X and O is centered in its square.
- The size of all strokes is 1 pixel.
- The title of the display frame is "TicTacToe Board". (On a lab machine, other parts of the frame title bar will be different, since I generated my screen shots on a Mac. The title text is the only thing you need.)
- Anti-aliasing need not be turned on, but it's OK if it is.
- The program is invoked from the command line like this

```
java TicTacToe
```

Scoring out of 100% is as follows:

- 30% for the board itself
- 10% each for proper placement of the X's and O's
- -20% for violation of the 50-line-per-method rule

The idea behind the -20 point deduction is to ensure that your use more than just one monolithic `paintComponent method`. Using helper paint methods will in fact make the program simpler and quicker to write.

You do NOT have to provide any documentation for any part of your program. You do NOT have to do any error handling of any form.

To submit, run the following command on vogon:

```
handin gfisher mdtm TicTacToe.java TicTacToeComponent.java
```