Name: **SOLUTION**

Student ID:

Signature:

Langara College

CPSC 1181 – Midterm #1 (Weeks 1-4) – Section 002

June 6, 2017, 8:30am

Instructor: Jeremy Hilliker

Duration 1h45m (no extensions)

- Permitted aids: 1 single-sided letter/a4 reference sheet, hand written by the student taking the exam; no other aids
  - Reference sheet must be submitted with your exam (failure to do so will result in a penalty)
  - If you do not have a reference sheet, you must alert the invigilator before the first half of the exam expires.
  - You may remove the Appendices from the exam, but they too must be handed in
- Turn off all electronic devices
- Store all personal belongings out of reach
  - Ask the invigilator for permission before retrieving any personal belongings
- Strictly no speaking to other students during the exam
- The work submitted for this exam must be entirely your own
- Answer all questions in the provided spaces
  - Use backs of facing pages if more space is needed
- The exam is over when announced by the invigilator
- Print your name and student ID on the cover sheet of this exam
- Follow best programming practices to receive full marks
  - You may omit comments (except where noted)

Q	Max	Awarded
1	10	
2	10	
3	9	
4 (pg 5)	8	
4 (pg 6)	12	
4 (pg 7)	9	
4 (pg 8)	7	
5	6	
Total	71	

[10] Q1 Using Objects

Answer the following questions with respect to the classes “Checker” and CheckerDriver in the Appendix 1.

List Checker’s instance variable(s).

number [not “count”]

List all of the local variables declared in the “main” method.

a, b, c [“args” okay]

How many Checker objects are created by the “main” method of CheckerDriver?

5

How many of those objects remain reachable before executing the final line of the “main”?

2

How many objects are orphaned before executing the final line of “main”?

3

What objects, if any, are used by “main” that it did not create?

System.out. & The Strings returned by Checker’s toString() method

What is the output of running the “main” method?

1 1

1 1

5 5

2 5

[10] Q2 GUI

Complete the following GUI code.

```
import java.awt.*;
/** A diamond figure.  A quadrilateral with equal sides and equal
angles. A square rotated 45 degrees. See Appendix 2.
*/
public class Diamond {
  private final int x;
  private final int y;
  private final int size;
  private final Color color;

  /**  Creates a Diamond to be drawn with the given brush color.
    @param anX the x coordinate of the top left of the bounding box of the diamond
    @param anY the y coordinate of the top left of the bounding box of the diamond
    @param aSize the height and width of the bounding box containing the diamond
    @param aColor the colour of the diamond.
                  If null, does not change colour when drawing.  */
  public Diamond(int anX, int anY, int aSize, Color aColor) {
x = anX;
y = anY;
size = aSize;
color = aColor;

  }
  public void draw(Graphics2D g) {

    //easy way
    // N, E, S, W
    int[] xS = new int[] {x+size/2, x+size, x+size/2, x };
    int[] yS = new int[] { y, y+size/2, y+size, y+size/2 };
    g.setColor(color);
    g.drawPolygon(xS, yS, xS.length); // using drawLine is okay


    /*
    // hard way
    g.translate(x+size/2, y);
    g.rotate(Math.PI/4);
    g.setColor(color);
    int adj2 = (size/2) * (size/2);
    int sideLength = (int) Math.sqrt(adj2 + adj2);

    g.drawRect(0,0, sideLength, sideLength);
    */


  }
}
```

[9] Q3 GUI

Complete the following GUI code. Fill in the blanks.

```
import javax.swing.*;
import java.awt.*;

public class DiamondComp extends JComponent {

  /**  Draws a Diamond so that it fills the height or width (whichever is lesser)
  of the component without becoming distorted (all angles remain right angles).
  If the diamond is restricted [limited, bound] by its WIDTH, it is drawn in blue.
  If the diamond is restricted [limited, bound] by its HEIGHT, it is drawn in red.
  See Appendix 3 for examples.
  */
  public void paint(Graphics g) {
     // get height & width of component
     final int w = getWidth();
     final int h = getHeight();




     Diamond d;
     if(w > h) {
       d = new Diamond(w/2 - h/2, 0, h, Color.RED);
     } else {
       d = new Diamond(0, h/2 - w/2, w, Color.BLUE);
     }

     d.draw((Graphics2D) g);









  }
  public static void main(String[] args) {
    JFrame frame = new JFrame();
    frame.setSize(500, 500);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.add(new DiamondComp());
    frame.setVisible(true);
  }
}
```

[30] Q4 Making Objects [8 this page]

Implement the following class with respect to the "Robot" class in Appendix 4. Fill in the blanks.

```
import java.util.*;
public class RobotGarage {
// declare a collection called "robots"
```

```
private final HashSet<Robot> robots;
```

```
  public RobotGarage() {
```

```
robots = new HashSet<Robot>();
```

```
  }

  public void add(Robot r) { robots.add(r); }
  public void remove(Robot r) { robots.remove(r); }
  public int size() { return robots.size(); }

  /** Determine if the garage contains a robot with the given ID.
       @param id the id to serach for
       @return true iff the garage contains a robot with the given id. */
  public boolean contains(int id) {
```

```
    for(Robot r : robots) {
      if(r.getID() == id) {
        return true;
      }
    }
    return false;
```

```
  }
```

```
/** Determine if all of the given robots are contained in the garage.
     @param ids the ids to search for in the garage
     @return true iff the garage contains robots matching each id */
public boolean contains(int... ids) {
```

```
   for(int id : ids) {
     if(!contains(id)) {
       return false;
     }
   }
   return true;
```

```
}
```

```
/** Gets the robot with the highest odometer. Choose one arbitrarily if there
     is a tie. Null if the garage is empty.
     @return the Robot with the highest odometer, or null if garage is empty. */
public Robot getHighestOdometer() {
```

```
   Robot highest = null;
   for(Robot r : robots) {
     if(highest == null ||
       r.getOdometer() > highest.getOdometer()) {
       highest = r;
     }
   }
   return highest;
```

```
}
```

… Q4 continued [9 this page]

```
/** Removes all the robots from the garage that are out of warranty.
     @return true if the garage was changed. */
public boolean removeOutOfWaranty() {

    boolean changed = false;
    Iterator<Robot> it = robots.iterator();
    while(it.hasNext()) {
      Robot r = it.next();
      if(!r.isUnderWarranty()) {
        it.remove();
        changed = true;
      }
    }
    return changed;

}
```

… Q4 continued [7 this page]

```
  /** Finds the number of unique robots in this garage and another garage.
       Ie: returns the size of the union of the two garages.
       Ie: size( A ∪ B = { x : x ∈ A | x ∈ B} )
       Eg: {1,2,3}.numUnique({2,3,4}) -> 4
       Eg: size( {1,2,3} ∪ {2,3,4} = {1,2,3,4} ) -> 4
       Eg: size( {1,2,3} ∪ {  } = {1,2,3} ) -> 3
       @param the other garage to compare with
       @return number of unique Robots in the two garages. */
  public int numUniqueRobots(RobotGarage o) {
```

```
    int unique = this.robots.size();
    for(Robots r : o.robots) {
      if(!this.robots.contains(r)) {
        unique++;
      }
    }
    return unique;




/*  // opposite of above
    int unique = this.robots.size() + o.robots.size();
    for(Robots r : o.robots) {
      if(this.robots.contains(r)){
        unique--;
      }
    }
    return unique;



    // okay way
    HashSet<Robot> allRobots = new HashSet<Robot>();
    allRobots.addAll(this.robots);
    allRobots.addAll(o.robots);
    return allRobots.size();

*/
```

```
  }
}
```

[6] Q5 Testing

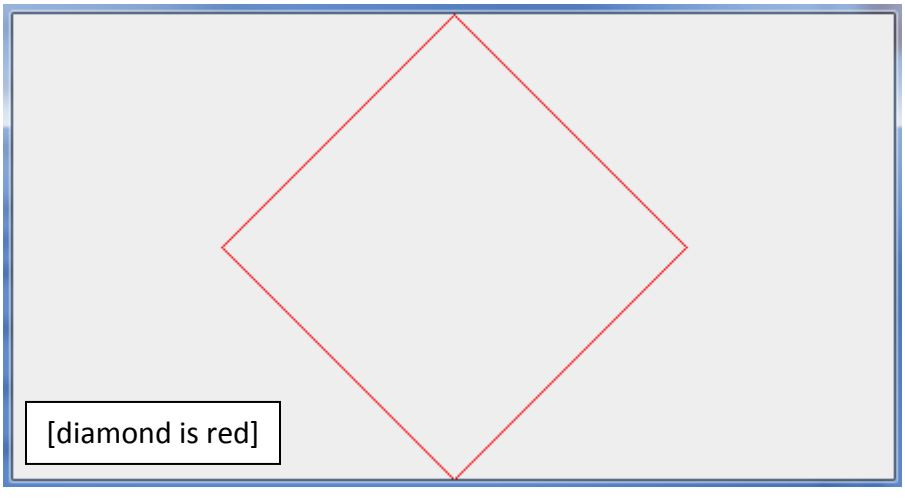
Write test cases for the "getHighestOdometer()" method of Q4. Label your tests.

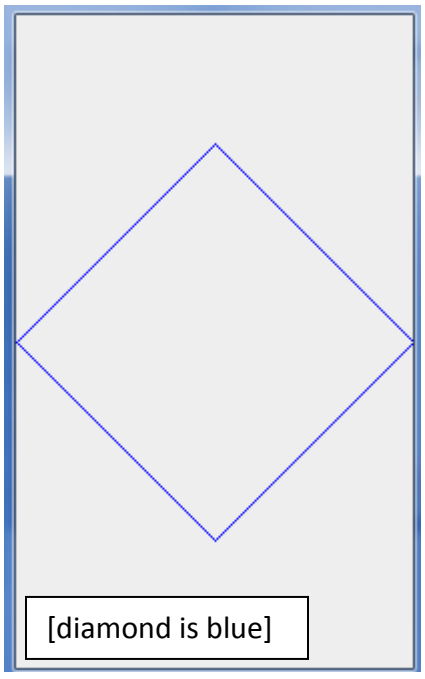
```
public class RobotGarageTester {
  public static void main(String[] args) {
    // test getHighestOdometer()
    RobotGarage g = new RobotGarage();

    // special, empty
    assert null == g.getHighestOdometer();

    // boundary, 1 robot
    Robot r1 = new Robot();
    g.add(r1);
    assert r1 == g.getHighestOdometer();

    // typical, multiple robot
    Robot r2 = new Robot();
    Robot r3 = new Robot();
    r2.move(10);
    r3.move(1);
    g.add(r2);
    g.add(r3);
    assert r2 == g.getHighestOdometer();

    // typical, after remove
    g.remove(r2);
    assert r3 == g.getHighestOdometer();

// one of:
    // typical, Robot moves after add
    g.add(r2);
    r3.move(100);
    assert r3 == g.getHighestOdometer();

    // a tie
    g = new RobotGarage();
    r1 = new Robot();
    r2 = new Robot();
    g.add(r1); g.add(r2);
    assert r1 == g.getHighestOdometer()
        || r2 == g.getHighestOdometer();



    System.err.println("*** PASS ***");
  }
}
```

Appendix 1

```
public class Checker {
  private static int count = 0;
  private int number = 0;

  public Checker(){
    count++;
    number = count;
  }

  public String toString() {
    return number + " " + count;
  }
}
public class CheckerDriver {
  public static void main(String[] args) {
    Checker a = new Checker();
    Checker b = a;
    Checker c;
    System.out.println(a.toString());
    System.out.println(b.toString());

    a = new Checker();
    b = a;
    a = new Checker();
    a = new Checker();
    a = new Checker();
    System.out.println(a.toString());
    System.out.println(b.toString());
  }
}
```

Appendix 2 & 3





Appendix 4

```
public class Robot {
  private final static int WARANTY = 100000;
  private static int nextID = 101;

  private final int id;
  private int odometer;

  /** Creates a Robot with a unique ID and odometer set to 0.
  */
  public Robot() {
    id = nextID++;
  }

  /** Moves the Robot. Odometer' = odometer + howFar
    @param howFar the distance to move the robot.
  */
  public void move(int howFar) {
    odometer += howFar;
  }

  /** Gets the robot's unique ID.
    @return the robot's unique ID.
  */
  public int getID() {
    return id;
  }

  /** Get the robot's current odometer.
    @return how far the robot has moved over its life.
  */
  public int getOdometer() {
    return odometer;
  }

  /** Determines if the robot is still under warranty.
    @return true if the robot is still under warranty, false otherwise
  */
  public boolean isUnderWarranty() {
    return odometer < WARANTY;
  }

  /** Determines if this robot is the same robot as another.
    @param o the other robot to compare to
    @return true if the other robot is the same as this one,
            False otherwise.
  */
  public boolean equals(Robot o) {
    return this.id == o.id;
  }
}
```

Appendix API

Graphics2D
Graphics create()
void dispose()
void draw(Shape s)
void drawLine(int x1, int y1, int x2, int y2)
void drawRect(int x, int y, int width, int height)
void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)
void drawPolygon(Polygon p)
void drawPolyline(int[] xPoints, int[] yPoints, int nPoints)
AffineTransform getTransform()
void rotate(double theta)
void rotate(double theta, double x, double y)
void scale(double sx, double sy)
void setColor(Color c)
void setStroke(Stroke s)
void setTransform(AffineTransform Tx)
void translate(int x, int y)

Rectangle
Rectangle(int x, int y, int width, int height)
void translate(int dx, int dy)

Polygon
Polygon()
Polygon(int[] xpoints, int[] ypoints, int npoints)
void addPoint(int x, int y)
void translate(int deltaX, int deltaY)

Collections
HashSet<E>()
ArrayList<E>()
void add(E e)
void addAll(Collection<? extends E> c)
void clear()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
boolean equals(Object o)
boolean isEmpty()
Iterator<E> iterator()
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
int size()

Iterator<E>
boolean hasNext()
E next()
void remove()