This exam is closed-book, closed-notes, no electronic devices allowed. The exception is the 
“sage page” on which you may have notes to consult during the exam. Answer questions on the 
pages of the exam. Do not unstaple the pages of this exam, nor should you attach any other pages 
to the exam. You are welcome to use the blank space of the exam for any scratch work.

Your work must be legible. Work that is difficult to read will receive no credit. Do not dwell over 
punctuation or exact syntax in code; however, be sure to indent your code to show its structure.

You must sign the pledge below for your exam to count. Any cheating will cause the students 
involved to receive an F for this course. Other action may be taken. If you need to leave the room 
for any reason prior to turning in your exam, you must give your exam and any electronic devices 
with a proctor.

You must fill in your identifying information correctly. Failure to do so is grounds for a zero on 
this exam. When you reach this point in the instructions, please give the instructor or one of the 
proctors a meaningful glance.

<table>
<thead>
<tr>
<th><strong>Print clearly</strong> the following information:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name (print clearly):</td>
</tr>
<tr>
<td>Student 6-digit ID (one digit per box):</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Your answers below tell us where to return your graded exam.</th>
</tr>
</thead>
<tbody>
<tr>
<td>What time do you arrive in studio/lab? (circle one) 11:30 1:00 2:30 4:00</td>
</tr>
<tr>
<td>Which Urbauer lab? (your best guess, circle one) 214 216 218 222</td>
</tr>
</tbody>
</table>

**Pledge:** On my honor, I have neither given nor received any unauthorized aid on this 
exam.

Signed: ________________________________

(Be sure you filled in your information in the box above!)
<table>
<thead>
<tr>
<th>Problem Number</th>
<th>Possible Points</th>
<th>Received Points</th>
<th>Grader Initials</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. (15 points)
Throughout this and other problems pertaining to our project, assume unless otherwise stated that we are playing the CSE131 game of Mastermind:

- 6 possible colors, encoded as 0, 1, 2, 3, 4, or 5
- 4 holes per row
- 12 allowable guesses
- No repeated colors in a code, but you can provide guesses with repeated colors
- The game ends if you supply the same guess more than once or if you run out of moves

In our lab 9 and project, we had the HonestFeedback object to evaluate a given guess’s merit against a secret code, which implemented the ProvidesFeedback interface, shown on the next page. Consider here another implementation of that interface, which we will call the HaterFeedback class, which behaves as follows:

- It fully implements the ProvidesFeedback interface.
- It judges every guess g as follows:
  - It always finds that g has no pegs in the same position as the solution.
  - It always finds that g has no colors in common with the solution.
  - It always determines that g is not the correct solution.

It does these things even if g is the solution!

(a) (2 points) Assuming you don’t know the solution, how many guesses would you have to supply to the HaterFeedback implementation to determine that it is lying to you?

(b) (3 points) Provide a specific example of such guesses below:
public interface ProvidesFeedback {

    /**
     * For the supplied Guess, how many peg ids occur in the correct position
     * with respect to the desired goal?
     * For example, if the goal is 0 2 3 1
     * and the guess is 0 1 3 4
     * then the result is 2, because the pegs with ids 0 and 3 are in exactly the
     * right position.
     * @param guess the Guess to be judged with respect to the goal
     * @return the number of peg ids in the correct position
     */
    public int numSamePosition(Guess guess);

    /**
     * Given the supplied Guess, how many peg ids are in common with the desired
     * goal and the guess?
     * For example, if the goal is 0 2 3 1
     * and the guess is 0 1 3 4
     * then the result is 3 because peg ids 0, 1, and 3 occur in both the goal and
     * the guess.
     * @param guess the Guess to be judged with respect to the goal
     * @return the number of peg ids in common with the goal and the guess
     */
    public int numIntersection(Guess guess);

    /**
     * Is the solution correct? This can be reduced from numSamePosition(guess)
     * returning an answer showing all pegs of the Guess are in the correct
     * position.
     * @param guess the supplied Guess to be judged
     * @return true iff the guess is completely correct
     */
    public boolean isSolution(Guess guess);
}

Continued on next page...
(c) (10 points) Below, write the HaterFeedback class as described above.

```java
//
// Fill this in
//
public class HaterFeedback implements ProvidesFeedback {
```
2. (20 points) Consider the following interface:

```java
public interface ProvidesArea {
    public double getArea();
}
```

In this problem you are given the beginnings of a `Rectangle` and `Circle` class. Your job is to finish them by:

- Completing the constructor (5 points)
- Completing the `getArea()` method (5 points)

Here are the “stories” related to the constructors you are provided for these objects:

- When a `Rectangle` is constructed it is provided the lower-left and upper-right points of the specified rectangle. These are supplied as two `ints` each: `llx` and `lly` denote the lower-left corner, and `urx` and `ury` denote the upper-right corner. Assume that these coordinates are supplied without error: the lower-left point is no higher or further to the right than the upper-left point. In other words, `llx ≤ urx` and `lly ≤ ury`.

- When a `Circle` is constructed, it is provided a center point and a radius. The center point is specified as two `ints`, `x` and `y`, and the radius is supplied as the `int r`.

Some notes:

- In the work that follows for this problem, you are not allowed to introduce any other instance variables.
- Recall that the area of a rectangle is the product of its length and width.
- Recall that the area of a circle is `Math.PI` times its radius squared.

Continued on next page...
(a) (10 points)

```java
public class Rectangle implements ProvidesArea {
    private final int llx, lly, urx, ury;

    public Rectangle(int llx, int lly, int urx, int ury) {

    }

    public double getArea() {

    }
}
```

(b) (10 points)

```java
public class Circle implements ProvidesArea {
    private final int x, y, r;

    public Circle(int x, int y, int r) {

    }

    public double getArea() {

    }
}
```
3. **(15 points)** This problem pertains to how objects are organized in memory. We will use only *ints*, as we did in the lecture material (videos) for module 7. As was the case there, we make the simplifying assumption that every *int* takes up a single location in memory, and that memory begins at address 1000. Memory and its contents are shown in the table below:

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
<th>Address</th>
<th>Contents</th>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>12</td>
<td>1006</td>
<td>-10</td>
<td>1012</td>
<td>4</td>
</tr>
<tr>
<td>1001</td>
<td>-14</td>
<td>1007</td>
<td>-12</td>
<td>1013</td>
<td>5</td>
</tr>
<tr>
<td>1002</td>
<td>-3</td>
<td>1008</td>
<td>2</td>
<td>1014</td>
<td>10</td>
</tr>
<tr>
<td>1003</td>
<td>0</td>
<td>1009</td>
<td>2</td>
<td>1015</td>
<td>11</td>
</tr>
<tr>
<td>1004</td>
<td>0</td>
<td>1010</td>
<td>8</td>
<td>1016</td>
<td>40</td>
</tr>
<tr>
<td>1005</td>
<td>4</td>
<td>1011</td>
<td>4</td>
<td>1017</td>
<td>4</td>
</tr>
</tbody>
</table>

The above table contains some *Rectangle* and *Circle* objects, whose definitions appeared in Problem 2. Based on object layout for those objects, answer the following questions.

(a) **What is the radius of the *Circle* whose center is at the origin (0,0)?**

(b) **At which address do you find a *Rectangle* that is actually a square?**

(c) **What is the area of a *Rectangle* if it is found at address 1008?**

(d) **What is the (approximate) area of a *Circle* if it is found at address 1008?**

(e) **Why would a *Rectangle* found at address 1004 be incorrectly specified?**

---

4. **(10 points)** For each of the scenarios below, place an X in the column whose associated abstract data type (ADT) is better suited for the scenario.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>List</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>The sequence of <em>Guesses</em> in a Mastermind game</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The colors of pegs in a <em>Guess</em>, from left to right</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The <em>Terms</em> of a <em>SparsePolynomial</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The steps performed to bake a cake</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The utensils (fork, spoon, measuring cup, etc.) needed to bake a cake</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5. (20 points) A song has a title, an artist, and a duration in seconds. It also keeps track of how many copies of the song have been purchased. A method `purchase()` should be included to indicate that one more copy of the song has been purchased.

Below (continue on the back of the exam if necessary), write the code for the `Song` class. Be sure to include the constructor, instance variables, and methods that are necessary based on the story. Do not worry about `hashCode` or `equals`.

```java
public class Song {
}
```
6. (20 points) In this problem you consider writing another method for the Rectangle class introduced in Problem 2. We are interested in computing a Rectangle that is the smallest such rectangle that contains two other Rectangles. A picture of this is shown below.

Given Rectangles r1 and r2, shown with solid lines, the smallest Rectangle that contains both of those is shown with dashed lines.

The method you are to write for class Rectangle from Problem 2 has the following signature:

```
public Rectangle containsBoth(Rectangle other)
```

(a) (5 points) Why does the method apparently take in one Rectangle instead of two?

Continued on next page...
(b) (10 points) Below, write a method that returns a new Rectangle that is the smallest rectangle to contain the two Rectangles of interest. You must not modify the instance variables of either Rectangle. It should be the case that

\texttt{r1.containsBoth(r2)}

returns a Rectangle with the same coordinates as

\texttt{r2.containsBoth(r1)}

\texttt{public Rectangle containsBoth(Rectangle other) { }

(c) (5 points) We would like two Rectangles to equal each other if they overlap each other exactly when drawn.

i. (2 points) Why can we not use \texttt{getArea()} to determine equality?

ii. (3 points) On what instance variables should equality and hashCode be determined for a Rectangle?