CS 125 Final Exam Solutions

—SOLUTION SET—

16 Dec 2017

This three-hour final exam consists of six questions broken into three types:

1. four 20-point short-answer questions;
2. one 35-point medium answer question; and
3. one 65-point long answer question.

Point values assigned to each question are intended to help you budget your time—so you should work on a 10-point question for about 10 minutes. Answer each question as **clearly** and **succinctly** as possible. Draw pictures or diagrams if they help. If you think that a question is unclear or ambiguous, state any assumptions you are making as part of your answer. **Please write clearly.** You may lose points if we cannot read your answer.

No aids of any kind are permitted. And you may not leave the exam room with any pages from this exam or written notes. Do not detach any pages from this exam.

Statistics:

- **613** students took this exam.
- **129** was the median score.
- **122.8** was the average score.
- **31.73** was the standard deviation of the scores.

1. Sorting and Searching (20 Points)

(a) **8 points** We have studied four sorting algorithms this semester—selection sort, insertion sort, mergesort, and quicksort. Fill in the table below with the *best* and *worst* case runtime for each algorithm using $\mathcal{O}$ notation[1] (1 points each).

Solution

Algorithm	Best Case	Worst Case
Selection Sort	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$
Insertion Sort	$\mathcal{O}(N)$	$\mathcal{O}(N^2)$
Mergesort	$\mathcal{O}(N\ log\ N)$	$\mathcal{O}(N\ log\ N)$
Quicksort	$\mathcal{O}(N\ log\ N)$	$\mathcal{O}(N^2)$

Question 1a Statistics:
- **8** was the median score.
- **7.16** was the average score.
- **1.61** was the standard deviation of the scores.

(b) **4 points** Using insertion sort I sorted (10^6) unordered email addresses in 4.0 seconds. Ignoring possible memory limitations of my machine, approximately how long will it take to sort (10^8) unordered email addresses?

◯ $<$ 3 minutes ◯ 3–10 minutes ◯ 11–59 minutes √ **1–23 hours**
◯ 24–72 hours ◯ $>$ 72 hours

Solution: Insertion sort on random data scales as N^2. The approximate running time will be 4000 seconds, slightly more than one hour. So 1–23 hours is the correct choice.
Question 1b Statistics:
- **0** was the median score.
- **1.57** was the average score.
- **1.95** was the standard deviation of the scores.

[1] What we have called "big O notation".

(c) **8 points** Each week the agro-computational "compu-corn-icans" monks write the date and crop sales onto a new heavy clay tablet which they archive with others along a wall edge. They want to re-sort their tablets by lifting and moving these so that the tablets are now ordered by increasing number.
Which CS 125 sorting algorithm would you recommend if comparing tablets is easy but moving tablets is hard? Justify your answer.

Solution: Selection sort minimize the number of swaps, which are the costly operations in this example.

Question 1c Rubric:

1. **4 points** for choosing selection sort as the correct algorithm
2. **4 points** for identifying that the swaps as the expensive operation for this particular sort

Question 1c Statistics:

- **8** was the median score.
- **5.45** was the average score.
- **3.63** was the standard deviation of the scores.

2. Algorithm Analysis (20 Points)

For each snippet below list the *worst case* runtime using $\mathcal{O}$ notation (4 points each).

```
void foo1(final int N) {
    int i = N * N;
    int j = N / 2;
    while (i > 0) {
        i = i - N;
        j = j + (int) Math.sqrt(640000);
    }
}
```

(a) **4 points** `foo1` $\mathcal{O}(N)$

```
/*
 * The length of data is N. Initially called with idx = 1.
 */
void foo2(final int[] data, final int idx) {
    if (idx >= data.length) {
        return;
    }
    foo2(data, 2 * idx);
```

```
        data[idx] = idx * idx + data[idx - 1];
    }
```

(b) **4 points** `foo2` $\mathcal{O}(log\ N)$

```
    /*
     * The length of data is N.
     */
    void foo3(final double[] data, final double threshold) {
        for (int idx = 1; idx < data.length; idx++) {
            if (data[idx] > threshold) {
                for (int j = 0; j <= idx; j++) {
                    data[j]--;
                }
            }
        }
    }
```

(c) **4 points** `foo3` $\mathcal{O}(N^2)$

```
    /*
     * mysteryFoo(N) scales as O(N log N)
     */
    void foo4(final int N) {
        for (int idx = N; idx > 0; idx--) {
            mysteryFoo(N);
        }
    }
```

(d) **4 points** `foo4` $\mathcal{O}(N^2\ log\ N)$

```
    int foo5(final int N) {
        if (N < 2) {
            return N;
        }
        return foo5(N % 2) + foo5(N / 2);
    }
```

(e) **4 points** `foo5` $\mathcal{O}(log\ N)$

(a) Chuchu in full color.

(b) Chuchu posterized in 10 colors.

Figure 1: Example of posterization.

Question 2 Statistics:

- **16** was the median score.
- **14.48** was the average score.
- **5.84** was the standard deviation of the scores.

3. Posterization (20 Points)

While photographs can contain many colors, certain situations require reducing the number of colors in an image to a much small number. For example, certain T-shirt printers print only one color at a time, making the cost scale roughly as $\mathcal{O}(N)$ with the number of colors. This process is sometimes referred to as *posterization*, since apparently at one point it was used to create posters.

To answer this question, describe an algorithm to posterize an image into N different RGB colors which are provided as inputs. The output image should consist *only* of pixels in the given set. **Write your answer for Question 3 on the next page.**

(a) **5 points** First, provide a written description of your algorithm in English. You will need to consider and carefully define what it means for two colors to be similar. We will accept multiple definitions of color similarity, but ensure that yours is well-motivated and clearly explained.

Question 3a Solution:
Here is one possible answer:

> To posterize an image, we must loop over each pixel in the image. For each pixel, we must determine which of the input colors its color is closest to. We define the distance between two pixels as the sum of the absolute values of the differences of each pixel channel value. This means that a color is always closest to itself, and similar hues of the same color are also close to it—so our definition seems reasonable. For each of our input colors, we compute the distance from it to the current pixel, and

save the input color that is currently the closest. When our loop completes, we replace that pixel with the closest input color.

Question 3a Rubric:

1. **5 points** for a reasonable explanation of your approach
2. **3 points** for a confusing or incorrect explanation, or one that omits certain key details
3. **0 points** for a blank or completely incomprehensible answer

Question 3a Statistics:

- **5** was the median score.
- **4.37** was the average score.
- **1.33** was the standard deviation of the scores.

(b) **10 points** Next, provide an implementation of your algorithm in either Java or pseudocode. In contrast to your English description for the first part, your answer here should be able to be translated to valid Java code with minimal additional work. Feel free to use shorthand for extracting color channels from pixels, rather than the bit operations that you used for MP4. For example, you can write `pixel.red` to refer to the red channel value of `pixel`.

Question 3b Solution:
Here's an example in nearly-correct Java code that follows the written explanation above. Obviously we asked for pseudocode, so you didn't need to write something this detailed. See the rubric below for grading details.

```
1
2  /**
3   * Posterize an image.
4   *
5   * @param image two-dimensional array of pixels as Color objects to posterize
6   * @param colors one-dimensional array of colors to convert array colors to
7   * @returns the posterized image
8   */
9  Color[][] posterize(Color[][] image, Color ...colors) {
10
11     // Should do some error checking on the input image, but this is pseudocode
12     Color[][] posterizedImage = new Color[image.length][image[0].length];
13
14     for (int x = 0; x < image.length; x++) {
15         for (int y = 0; y < image[x].length; y++) {
16
17             Color closestColor = null;
18             int closestDistance = 0;
19             for (Color option: colors) {
20                 int optionDistance = Math.abs(image[x][y].red - option.red) +
21                                      Math.abs(image[x][y].green - option.green) +
22                                      Math.abs(image[x][y].blue - option.blue);
23                 if (closestColor == null || thisDistance < closestDistance) {
```

```
24                      closestColor = option;
25                      closestDistance = optionDistance;
26                  }
27              }
28              posterizedImage[x][y] = closestColor;
29          }
30      }
31
32      return posterizedImage;
33  }
```

Question 3b Rubric:
Grading for this question focused on two components of your pseudocode implementation: the distance calculation and the loop structure.

1. **Distance calculation**
   a) **5 points** for a reasonable implementation of pixel or color distance
   b) **3 points** for minor mistakes
   c) **1 point** for something completely broken
   d) **0 points** for no distance calculation
2. **Loop structure**
   a) **5 points** for a correct loop.
   b) **3 points** for an incorrect loop.
   c) **0 points** for no loop.

Question 3b Statistics:

- **4** was the median score.
- **4.39** was the average score.
- **3.27** was the standard deviation of the scores.

(c) **5 points** Finally, indicate the running time of your algorithm using $\mathcal{O}$ notation, where N is the number of pixels (width $\times$ height). If your algorithm could be made more efficient, describe how. If your algorithm could *not* be made more efficient, argue why not.

Question 3c Solution:
At least for a reasonable number of colors, what dominates here is the need to touch every pixel in the image. However, if the number of colors gets extremely large, that part of the loop can start to take over.

More formally, given k colors and a square n by n image, the runtime would be $\mathcal{O}(k \times n \times n)$. If $k >> n$, then it is essentially $\mathcal{O}(k)$. But if $n > k$, it is essentially $\mathcal{O}(n^2)$.

The problem didn't ask you to consider large numbers of input colors, since the idea behind posterization was to reduce to a smaller number of colors. So anything that pointed out that you had to touch every pixel was fine.G

Question 3c Rubric:

1. **5 points** for something that scales with the number of pixels in the image
2. **3 points** for a confusing or incorrect explanation or one that doesn't scale with the number of pixels
3. **0 points** for a blank or completely incomprehensible answer

Question 3c Statistics:

- **5** was the median score.
- **4.20** was the average score.
- **1.70** was the standard deviation of the scores.

4. `checkstyle` (20 Points)

```
1  public class Whatevs {
2      private static final String empty = "";
3
4      public static void doIt(String[] wawa) {
5
6          String a = empty, b = empty;
7          for (String c : wawa)
8          {
9              if (c.length() > 1024) {
10                 continue;
11             }
12             if (myMethod(b, c)) a = b = c;
13             else if (!myMethod(c, b))
14             a = a.concat("\n").concat(c);
15         }
16         // ready to print now... whee...
17         System.out.println(a);
18     }
19
20     static boolean myMethod(String a, String b) {
21         // this is definitely going to work
22         return a.length() > b.length() ? true : false;
23     }
24 }
```

(a) **10 points** First, identify *five* style problems with the code above (2 points per problem.) Assume we are using the same Sun Java style guidelines that we have been using all semester on the MPs, and apply the same guidance about code quality that we have repeated on MPs and in labs.

Question 4a Solution:
There were many, many problems with this code snippet. First, the ones that `checkstyle` would have complained about:

Line	Problem
1	Missing a Javadoc comment.
2	Missing a Javadoc comment.
4	Missing a Javadoc comment.
4	Parameter wawa should be final.
8	{ at column 5 should be on the previous line.
9	`1024` is a magic number.
12	`if` construct must use {'s.
12	Inner assignments should be avoided.
13	`if` construct must use {'s.
20	Missing a Javadoc comment.
20	Parameter `a` should be final.
20	Parameter `b` should be final.
22	Avoid inline conditionals.

Next, the other style problems that we've discussed this semester:

Line	Problem
1	`Whatevs` is not a descriptive name for this class.
2	There is no need to store an empty string as a private class variable.
4	`doIt` is not a descriptive name for this method.
4	`wawa` is not a descriptive name for this method paramater.
6	`a` and `b` are not descriptive names for these local variables.
7	`c` is not a descriptive name for this local variable.
9	`1024` is an arbitrary threshold and should be commented.
16	This comment is obvious and unnecessary. (And stupid.)
20	`myMethod` is not an appropriate name for this method.
21	Another spurious and unnecessary comment.
22	There is no need for this inline conditional, since it just returns the value of the check. Actually this entire method is not needed.

Question 4a Rubric:
As embedded in the question: **2 points** per problem correctly identified.
Question 4a Statistics:

- **8** was the median score.
- **8.11** was the average score.
- **2.37** was the standard deviation of the scores.

(b) **5 points** Next, rewrite the code to fix these problems. Your code should pass the `checkstyle` tests, and be concise and readable. Note that you *should not change the implementation* of the function—just fix the style errors.

Question 4b Solution:

Here is one possible approach. It's fairly maximal in that it cleans up the helper function by removing it. You didn't necessarily need to go this far, although you should have renamed and simplified helper function and provided Javadoc comments for it if you kept it.

```
1  /**
2   * A class that prints the longest lines from a multiline text.
3   */
4  public class LongestLines {
5
6      /** Maximum line length that we will process. Lines longer are ignored. */
7      public final static int MAX_LINE_LENGTH = 1024;
8
9      /*
10      * Print the longest lines from a multiline input text.
11      *
12      * Note that lines with a length of over MAX_LINE_LENGTH are ignored.
13      *
14      * @param input multiline text to pull longest lines from
15      */
16     public static void printLongestLines(final String[] input) {
17
18         String longestLine = "";
19         String longestLines = "";
20
21         for (String currentString : input) {
22             if (currentString.length() > MAX_LINE_LENGTH) {
23                 continue;
24             }
25             if (currentString.length() > longestLine.length()) {
26               longestLine = currentString;
27               longestLines = currentString;
28             } else if (currentString.length() == longestLine.length()) {
29               longestLines = longestLines.concat("\n").concat(currentString);
30             }
31         }
32         System.out.println(longestLines);
33     }
34 }
```

Question 4b Rubric:

1. **4 points** for fixing all five `checkstyle` errors that you identified in the previous part of your answer
2. **1 point** for fixing *all* `checkstyle` and other code quality errors

Question 4b Statistics:

- **4** was the median score.
- **3.25** was the average score.
- **1.23** was the standard deviation of the scores.

(c) **5 points** Finally, in a few sentences explain why style guidelines are important and not just super annoying.

Question 4c Solution:
Style guidelines make your code more readable. It also makes it easier with a group and contribute to large software projects, since every team members code looks the same. Past what `checkstyle` can check for, good variable naming conventions make it much easier for other people to read and understand your code. And excellent documentation is required whenever you are working with others—nobody will ever use your code without it!

Question 4c Rubric:
5 points for anything reasonable that didn't boil down to "because the CS 125 instructors told us so".

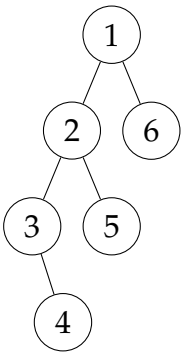
Question 4b Statistics:

- **5** was the median score.
- **4.96** was the average score.
- **0.45** was the standard deviation of the scores.

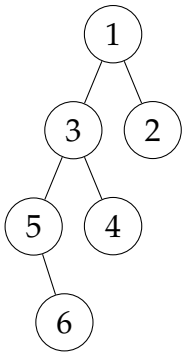
5. Tree Traversal (35 Points)

```
1  public class Tree {
2
3      /** Current node's parent. May be null if I'm the root of the tree. */
4      private Tree parent;
5
6      /** Current node's left child, or null if none. */
7      private Tree left;
8
9      /** Current node's right child, or null if none. */
10     private Tree right;
11
12     /** Current node's value. */
13     private int value;
14 }
```

Binary trees can be traversed in several different ways. A tree traversal visits every node in the tree in some order. One approach is known as *depth-first* traversal (DFT). It explores as far as possible along each branch in the tree before backtracking.



(2a) Visit order for DFT that visits left nodes first.



(2b) Visit order for DFT that visits right nodes first.

For example, assuming that a DFT visited all nodes in the following tree, they would be visited in the labeled order shown in Figure 2a. Note that these are not the *values* of the nodes—rather the order in which they are visited. This implementation visits left nodes first, but you could also visit right nodes first, at which point the order would be as shown in Figure 2b.

(a) **10 points** As a warm up, first, using our `Tree` data structure from MP6, write a recursive implementation of depth-first traversal. Descend to left nodes before right nodes, and print out the value of each node as you visit it.
Write a valid Java function to answer this question. You can assume that it would be run as an instance method of the `Tree` class defined above, and so can access its private instance variables. Use the instance that it is called on as the root to start your traversal.

Question 5a Solution:
This was fairly similar to what you worked on for MP5:

```
1  /**
2   * Recursive depth-first traversal.
3   */
4  public void recursiveDFT() {
5      System.out.println(value);
6      if (left != null) {
7          left.recursiveDFT();
8      }
9      if (right != null) {
10         right.recursiveDFT();
11     }
12 }
```

Question 5a Rubric:

1. **10 points** for a clear and correct implementation
2. **8 points** for minor mistakes, such as incorrect traversal order
3. **6 points** for serious mistakes, such as a missing base case
4. **4 points** for a non-recursive implementation or something else that is completely incorrect

Question 5a Statistics:

- **10** was the median score.
- **7.76** was the average score.
- **2.98** was the standard deviation of the scores.

(b) **20 points** Now, while recursion is a powerful programming technique, anything that can be done *with* recursion can also be done *without* recursion. Your next task is to (re)implement your depth-first traversal algorithm without using recursion. (No: your code cannot simply call your recursive implementation above. That counts as using recursion!) Follow the same guidelines as the previous part of the question: write a valid Java function, assume it is defined as a instance method, and use the instance it is called on to start your traversal.

You will need a data structure that maintains a list of `Tree` elements. You can assume it has the following interface—but note that you may not need all of these methods:

```
1  /*
2   * Assume that first and last are valid initialized Tree objects.
3   */
4
5  // Create a new empty list of Trees.
6  TreeList list = new TreeList();
7
8  // Prints true
9  System.out.println(list.empty());
10
11 // Add a value to the front of the list.
12 list.unshift(first);
13
14 // Prints 1
15 System.out.println(list.size());
16
17 /*
18  * Remove and return the first value from the list.
19  * Returns null if the list is empty.
20  */
21 first = list.shift();
22
23 // Adds the value to the end of the list.
24 list.push(last);
25
26 /*
27  * Remove and return the last value in the list.
28  * Returns null if the list is empty.
29  */
30 last = list.pop();
```

Question 5b Solution:
Here is the iterative solution:

```java
/**
 * Iterative depth-first traversal.
 */
public void iterativeDFT() {
    TreeList<Tree> list = new TreeList<>();

    list.unshift(this);
    while (!list.empty()) {
        Tree current = list.shift();
        System.out.println(current.value);
        if (current.right != null) {
            list.unshift(current.right);
        }
        if (current.left != null) {
            list.unshift(current.left);
        }
    }
}
```

Question 5b Rubric:
Grading for this question was split into two parts: 10 points for the correct usage of the list data structure, and 10 points for the iterative logic.

1. For usage of the list data structure:
   a) **10 points** for completely correct usage
   b) **5 points** for something containing mistakes
   c) **1 point** for at least using the `TreeList` data structure
2. For the iterative logic:
   a) **10 points** for completely correct iterative logic.
   b) **8 points** for minor mistakes.
   c) **6 points** for serious mistakes like adding to the wrong end of the list.
   d) **4 points** for an attempt that could have been moving in the right direction.
   e) **1 point** for anything that the grader could understand.

Question 5b Statistics:
This was by far the hardest question on the exam.

- **2** was the median score.
- **4.06** was the average score.
- **5.23** was the standard deviation of the scores.

(c) **5 points** Finally, compare the operation of your recursive solution with your non-recursive solution. Specifically, you might want to describe what plays the role of the list data structure for the recursive implementation? Use the similarity to explain how anything that can be done with recursion can also be done without recursion.

Question 5c Solution:
The key insight here is that the list data structure functions almost exactly like the recursive call stack. When you make a recursive call, you push that function's context

onto the call stack and begin executing immediately. When you are done, you move on to the next function on the stack.
The iterative solution works similarly, except it uses an explicit stack, rather than the implicit function call stack. This is why it looks so similar. And this approach can be extended to reimplement *any* recursive function.

Question 5c Rubric:

1. **5 points** for identifying that the list functions like the recursive call stack
2. **3 points** for an incorrect but decent attempt

Question 5c Statistics:

- **3** was the median score.
- **1.89** was the average score.
- **1.91** was the standard deviation of the scores.

6. Finding Friends (65 Points)

Write a Java class to track the location of your friends. It should allow you and your friends to move around and discover nearby friends. You can assume that this class only stores *your* friends, so every created Java friend object is a friend of yours. All data required by the problem should be stored by your new class, not somewhere else.

(a) **5 points** Your class should model your friends' name and location attributes. You should represent a friend's location as integer coordinates in two-dimensional space. Once created, a friend's name cannot change, but their position can.

(b) **10 points** You should provide at least two ways to create new friends (5 points each). One way allows you to create a new friend with a given name and location. The second allows you to create a new friend with a given name and at the same location as another friend.

Your friend class should enable the following functionality:

(c) **5 points** Friends should be able to update their location and move from place to place.

(d) **5 points** You consider two friends as equal if they are in the same location, even if their names are different.

(e) **10 points** You should be able to determine which friend is closest to you. You can compute distance as $d = Math.sqrt(x * x + y * y)$. Be sure to cast values appropriately to ensure that the return value is a floating point type.

(f) **10 points** You should be able to print out a list of all your friends and their current locations. You should implement this in part by utilizing the standard Java method allowing each friend to print their own name and location. However, this is not sufficient to print all of your friends.

(g) **20 points** Finally, provide a small example showing your class in action:

- Create several friends at different locations.
- Then they should all move to a new location.
- Create a new friend at the same location as one of your current friends.
- Now print all of their locations.
- Then, move yourself to the location of your closest friend.
- Verify that you and your friend are, at that point, equal to each other.

Write valid Java code for all your answers to this question. You will need to split your class definition across the pages that follow. Document your code carefully, and provide explanation as needed. However, you do not need to write your documentation in valid Javadoc format—although you can if you want.

Most of the points assigned to this problem will be for correctness, not for style. You do not need to worry about strictly following the `checkstyle` coding conventions. However, you should write clear and concise code, commented where needed.

Solution for Question 6

Solution:

The entire solution is long, so let's break it into parts.

Here's a snippet showing instance variables (Question 6a) and both constructors (Question 6b):

```
1  /**
2   * A class that tracks the locations of your friends.
3   */
4  public class Friend {
5      /** Our friend's name. */
6      private String name;
7
8      /** Our friend's X coordinate. */
9      private int x;
10
11     /** Our friend's X coordinate. */
12     private int y;
13
14     /**
15      * Create a new friend with the specified properties.
16      *
17      * @param setName our new friend's name
18      * @param setX our new friend's X coordinate
19      * @param setY our new friend's Y cooordinate
20      */
21     public Friend(final String setName, final int setX, final int setY) {
```

```
22          super();
23          name = setName;
24          x = setX;
25          y = setY;
26          for (int i = 0; i < MAX_FRIENDS; i++) {
27              if (ourFriends[i] == null) {
28                  ourFriends[i] = this;
29              }
30          }
31      }
32
33      /**
34       * Create a new friend with a given name but the same position as another friend.
35       *
36       * @param setName our new friend's name
37       * @param anotherFriend the existing friend who's location we should copy
38       */
39      public Friend(final String setName, final Friend anotherFriend) {
40          this(setName, anotherFriend.getX(), anotherFriend.getY());
41      }
42  }
```

Question 6a Rubric:

1. **3 points** for correctly storing the location, *or*
2. **2 points** for incorrect integer coordinates, *plus*
3. **1 point** for storing the `name` as a `String`, *plus*
4. **1 point** for marking `name` as private

Question 6a Statistics:

- **5** was the median score.
- **4.54** was the average score.
- **1.15** was the standard deviation of the scores.

Question 6b Rubric:

Grading for this question was split into two parts: 5 points for the main constructor and 5 points for the copy constructor.

1. For the main constructor:
   a) **5 points** for a complete and correct implementation
   b) **3 points** for an incorrect implementation, such as one that does not set every field
2. For the copy constructor:
   a) **5 points** for a complete and correct implementation
   b) **3 points** for an incorrect implementation

And for both, **-1 point** if you forgot to add the newly-created `Friend` to the friends list.

Question 6b Statistics:

- **9** was the median score.
- **8.38** was the average score.
- **2.60** was the standard deviation of the scores.

Here's a snippet showing location updates (Question 6c) and equality (Question 6d):

```
1  /**
2   * Get our current X coordinate.
3   *
4   * @return our friend's current X coordinate.
5   */
6  public int getX() {
7      return x;
8  }
9
10 /**
11  * Change our X coordinate.
12  *
13  * @param setX our friend's new X coordinate
14  */
15 public void setX(final int setX) {
16     x = setX;
17 }
18
19 /*
20  * Y setters and getters are equivalent, but omitted to save space.
21  */
22
23 /**
24  * Move a friend to a new location by changing both coordinates at once.
25  *
26  * @param setX our friend's new X coordinate
27  * @param setY our friend's new Y coordinate
28  */
29 public void move(final int setX, final int setY) {
30     this.setX(setX);
31     this.setY(setY);
32 }
33
34 /*
35  * (non-Javadoc)
36  * @see java.lang.Object#equals(java.lang.Object)
37  */
38 @Override
39 public final boolean equals(final Object obj) {
40     if (this == obj) {
41         return true;
42     }
43     if (obj == null) {
44         return false;
45     }
46     if (getClass() != obj.getClass()) {
47         return false;
48     }
49     Friend other = (Friend) obj;
50     if (x != other.x) {
51         return false;
52     }
53     if (y != other.y) {
54         return false;
55     }
56     return true;
57 }
```

Question 6c Rubric:

1. **5 points** for a correct location setter–either combined or separate
2. **3 points** for an incorrect location setter

Question 6c Statistics:

- **5** was the median score.
- **4.39** was the average score.
- **1.45** was the standard deviation of the scores.

Question 6d Rubric:

1. **5 points** for an `equals` method that correctly doesn't consider the `name` field
2. **3 points** for an incorrect `equals` method, such as one that compares all fields including `name`

Question 6d Statistics:

- **3** was the median score.
- **3.39** was the average score.
- **1.86** was the standard deviation of the scores.

Here's a snippet showing how to find your closest friend (Question 6e) and how to print all of your friends (Question 6f)

```
1  /**
2   * A class that tracks the locations of your friends.
3   */
4  public class Friend {
5      /** Nobody really needs more than this many friends. (Don't tell Facebook.) */
6      public static final int MAX_FRIENDS = 1024;
7
8      /** Array holding all of our friends. */
9      private static Friend[] ourFriends = new Friend[MAX_FRIENDS];
10
11     /** Us represented as a friend object. */
12     private static Friend me = new Friend("GWA", 0, 0);
13
14     /**
15      * Find your closest friend.
16      *
17      * @return our closest friend, or null if we have no friends
18      */
19     public static Friend closestFriend() {
20         Friend closestFriend = null;
21         double distanceToClosestFriend = 0.0;
22         for (Friend currentFriend : ourFriends) {
23             if (currentFriend != null) {
24                 double distance = Math.sqrt(Math.pow(((double) currentFriend.x
25                                                       - (double) Friend.me.x), 2)
26                                             + Math.pow(((double) currentFriend.y
27                                                         - (double) Friend.me.y), 2));
28                 if (closestFriend == null || distance < distanceToClosestFriend) {
29                     closestFriend = currentFriend;
30                     distanceToClosestFriend = distance;
```

```
31                  }
32              }
33          }
34          return closestFriend;
35      }
36
37      /**
38       * Print all of your friends information.
39       */
40      public static void printAllFriends() {
41          for (Friend currentFriend : ourFriends) {
42              if (currentFriend != null) {
43                  System.out.println(currentFriend);
44              }
45          }
46      }
47  }
```

Question 6e Rubric:

Grading for this question was split into three parts: 2 points for correctly storing your own location, 4 points for the distance calculation, and 4 points for the calculation loop.

1. For storing your own location:
   a) **2 points** for correctly storing your own location
2. For the distance calculation:
   a) **4 points** for a correct distance calculation
   b) **2 points** for minor errors like forgetting the `double` conversion
3. For the calculation loop:
   a) **4 points** for a complete and correct calculation loop
   b) **3 points** for something that is correct over an incorrect data structure or that returns the distance rather than the closest `Friend`
   c) **2 points** for major errors such as an incorrect loop

Question 6e Statistics:

- **8** was the median score.
- **7.14** was the average score.
- **2.60** was the standard deviation of the scores.

Question 6f Rubric:

Grading for this question was split into three parts: 5 points for the loop, 3 points for printing correctly, and 2 point for printing using `toString`.

1. For the loop:
   a) **5 points** for a loop that correctly accesses all friends
   b) **3 points** for a correct loop but over an incorrect data structure
   c) **2 points** for a working loop but created outside the class
2. For printing:
   a) **3 points** for correct printing using `System.out`

   b) **2 points** for minor printing errors
3. For using `toString`:
   a) **2 points** for printing using `toString`
   b) **1 points** for printing using some other method, or not printing all requested fields

Question 6f Statistics:

- **6** was the median score.
- **6.04** was the average score.
- **2.82** was the standard deviation of the scores.

And, finally, the requested example (Question 6g):

```
1  /**
2   * Example as required by the question.
3   *
4   * @param unused unused input arguments
5   */
6
7  public static void main(final String[] unused) {
8
9      Friend dog = new Friend("Chuchu", 10, 8);
10     Friend cat = new Friend("Xyz", 1, 1);
11     Friend sweetestOne = new Friend("Suzanna", 0, 0);
12
13     dog.move(1, 1);
14     cat.move(0, 0);
15     sweetestOne.move(2, 2);
16
17     Friend josh = new Friend("Josh", sweetestOne);
18
19     Friend.printAllFriends();
20
21     Friend.me.move(sweetestOne.getX(), sweetestOne.getY());
22
23     System.out.println(Friend.me.equals(sweetestOne));
24 }
```

Question 6g Rubric:

1. **4 points** for creating several friends, *plus*
2. **2 points** for moving them to a new location, *plus*
3. **2 points** for creating a new friend at a new location, *plus*
4. **4 points** for printing all locations, *plus*
5. **4 points** for moving to the location of the closest friend, *plus*
6. **4 points** for the equality test

Question 6g Statistics:

- **20** was the median score.
- **17.27** was the average score.
- **5.97** was the standard deviation of the scores.