Com S 227
Fall 2023
Assignment 4
300 points

Due Date: Friday, December 8, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm Dec 7)

NO LATE SUBMISSIONS! All work must be in Friday night.

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus, http://www.cs.iastate.edu/~cs227/syllabus.html **, for details.**
If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

Table of Contents

Introduction

For this assignment you will implement a number of classes for implementing some dice games that are known, with regional variations, as *Maxi Yatzy*, *Generala*, or (in the US) *Yahtzee*. Our implementation choices do not precisely match any of them, but are closest to *Maxi Yatzy*. The purposes of this assignment are:

- To use interfaces and inheritance in a realistic way
- To give you a chance to make some design decisions related to inheritance
- To give you more practice using arrays and ArrayLists

Summary of tasks

You will implement these two classes:

`Combination`
`MaxiYatzy`

plus, *at a minimum*, the following ten classes that (directly or indirectly) implement the **`ScoreBox`** interface:

`AllMatchScoreBox`
`CastleScoreBox`
`ChanceScoreBox`
`MatchTargetScoreBox`
`FullHouseScoreBox`
`LargeStraightScoreBox`
`NOfAKindScoreBox`
`ShortStraightScoreBox`
`TowerScoreBox`
`TwoPairsScoreBox`

Whoa, you say, that's a lot of code to write! Not really. If you are careful with the use of inheritance, you'll find that much of code can be reused between the classes. That is why we say "at a minimum": In addition to the classes listed above, you will implement whatever additional classes you decide are necessary in order to exploit inheritance to facilitate code reuse. See the discussion of score boxes below for more information.

All your code goes in the **`hw4`** package. The **`hw4`** package includes a basic skeleton of each class listed above. Each of the **`ScoreBox`** types includes a declaration of its required constructor. The **`Combination`** class skeleton includes a **`toString`** method that is fully implemented (though it will not work until you have the other methods written).

Using inheritance

Part of your grade for this assignment (e.g. 15-20%) will be based on how effectively you are able to use inheritance to implement everything with a minimum of code duplication. Here are some specific restrictions; see the later section " Requirements and guidelines regarding inheritance" for more details.

- You may not use **`public`**, **`protected`** or package-private variables. (*You can provide protected accessor methods or constructors when needed.*)

- You may not modify anything in the **`api`** package

Overview

Games such as Maxi Yatzy are normally played with five or six six-sided dice and a scorecard. One turn consists of rolling the dice up to three times and then recording the result in one of the designated boxes on the scorecard. A given combination of dice may, or may not, *satisfy* a given score box category, which in part determines how many points the player earns.

In our version, the number of dice, the range of numbers on the dice, the maximum number of rolls, and the categories on the scorecard will all be configurable.

If you are not at all familiar with this type of game, don't worry, it is not too complicated. Take a look at the Wikipedia page on Yahtzee for an overview, http://en.wikipedia.org/wiki/Yahtzee . (The main differences between Yahtzee and Maxi Yatzy are that Maxi Yatzy is typically played with six dice instead of five; there are additional score boxes not found in Yahtzee; and most interestingly, Maxi Yatzy allows you to save up unused rolls to use in later turns.)

The two key abstractions in the design are *combinations* and *score boxes*. The role of the game class **`MaxiYatzy`** is to manage the combination for the current turn, count rolls, and maintain two lists of score boxes.

Combinations

A combination, represented by the **`Combination`** class that you will implement, is a basically a list of integers representing the states of N dice (in the standard game there would be 6 numbers in the range 1 through 6). However, the dice are partitioned into two groups: *available* dice and *completed* dice. Initially, all dice are available. When the dice are "rolled" by calling the **`rollAvailableDice()`** method of **`MaxiYatzy`**, random values in the appropriate range are

generated for the available dice only; the completed values are fixed. If the maximum number of rolls has not yet been reached, the player can select some of the dice values to be moved to the completed list. Those values are then become fixed, and only the remaining available dice are re-rolled. Note that in this design, there is no such thing as an individual "die" object; a die value is just an integer in the list. All methods that return arrays containing die values just return the values in ascending order.

After the maximum number of rolls is reached, the game will automatically move all available dice to the completed list and the combination can no longer be modified. A new combination must be created for the next turn.

Note that clients do not normally create **`Combination`** objects directly; a combination is normally instantiated by invoking **`startTurn()`** on the **`MaxiYatzy`** object.

Combination example. Suppose we represent a combination as a string by listing first the available dice and then the completed dice in parentheses. For example, in a game with 5 dice, after the first roll we might see

```
2 3 3 4 6 ()
```

Depending on which score boxes you need to satisfy, you might select 2, 3, and 4 (perhaps in the hope of completing a *straight*). Now you have

```
3 6 (2 3 4)
```

The 6 and the other 3 are then replaced by random values on the next roll, but the 2, 3, and 4 you selected remain fixed. If (for example) you next rolled a 2 and a 5, you would have

```
2 5 (2 3 4)
```

At this point you could select the 5 (to make a "short straight", hoping that rerolling the 2 might give you a 1 or 6 to make a large straight). Now we have

```
2 (2 3 4 5)
```

If, as luck would have it, you then rolled a 4, you'd have

```
(2 3 4 4 5)
```

Assuming (as in most versions of the game) we only get three rolls, the 4 is automatically moved into the completed dice since we can't reroll it.

Score boxes

A *score box* represents one row of the score sheet. It stores the actual score for that category of the score sheet, and also stores the combination that was used to satisfy the score box. Most

importantly, a score box it contains the algorithms needed to a) determine whether a given combination *satisfies* the category, and b) determine what the *potential score* would be for a given combination.

There are many different possible score boxes, each with its own particular algorithms. For example, the traditional Maxi Yatzy game has a three-of-a-kind score box: a combination satisfies the score box if it contains any three numbers that are the same, and it is scored by summing those three numbers. The traditional Yahtzee game also has a "large straight" score box: a combination satisfies the category if it has 5 consecutive values, and it always receives a fixed score of 40.

This is where polymorphism becomes useful. The client using this code (e.g., think of the client as the text-based UI provided in the sample code) does not care about the details of what the score box categories are or how to calculate the scores. It just needs to be able to invoke methods on a score box to find out whether a given combination satisfies it, what the score would be, and to inform the score box when it was filled by a given combination.

Therefore, a score box is defined by an interface **`ScoreBox`**. This interface is already written and you should not modify it. See the javadoc for detailed descriptions of the methods. See the text-based UI to see how it is used from the client's point of view. In particular, you can see in the **`printCategories`** method where the UI just iterates over the categories and displays the potential score, or actual score, from each one.

The exact definition of each of the ten score box classes listed above can be found by reading the class javadoc, but here is a summary:

Score box name	How it's satisfied	How it's scored
`AllMatchScoreBox`	All dice have the same value	Fixed score, set in constructor
`CastleScoreBox`	At least three each of two different values	Sum of all dice
`ChanceScoreBox`	Any combination	Sum of all dice
`FullHouseScoreBox`	At least three of one value and two of a different value	Sum of all dice
`LargeStraightScoreBox`	All dice form a run of consecutive values	Fixed score, set in constructor
`NOfAKindScoreBox`	At least N dice with the same value	Sum of those N dice
`MatchTargetScoreBox`	Any combination	Sum of dice matching a specific target value, set in constructor
`ShortStraightScoreBox`	All but one of the dice form a run of consecutive values	Fixed score, set in constructor
`TowerScoreBox`	At least four of one value and two of a different value	Sum of all dice
`TwoPairsScoreBox`	At least two of one value and two of a different value	Sum of all dice

Note that a given combination may satisfy multiple score boxes. For example, if you had the combination (2, 2, 5, 5, 5, 6) it could satisfy any of:

Chance (25 points)
Match Target with target 5 (15 points)
Match Target with target 2 (4 points)

Full House (25 points)
N Of A Kind with N = 2 (10 points)
Two pairs (25 points)

Note that a combination may be used to *fill* a score box even if it does not meet the criteria for *satisfying* it; in that case the score is zero.

As listed above, there are ten concrete classes for you to write. Each of them must implement the **`ScoreBox`** interface. However, it need not do so *directly*: more likely, it would instead *extend* some other class that implements **`ScoreBox`**. If you just added the declaration **`implements ScoreBox`** to each of these classes and then proceeded to fill in all the required methods, you would find yourself writing a lot of the same code over and over again. Even though there are several different algorithms involved (e.g. checking for three of a kind is different from checking for a large straight), there is also a lot in common between the classes. You should carefully think about how to design an *inheritance hierarchy* so that you can minimize duplicated code between the classes. You might think about starting with an abstract class containing features that are common to *all* **`ScoreBox`** types, such as **`isFilled()`** or **`getDice()`**. There are additional opportunities for sharing code to think about too. Are there code similarities between "large straight" and "short straight" that you can exploit? Do you notice anything in common between FullHouse, TwoPairs, Castle, and Tower?

A portion of your grade on this assignment (roughly 15% to 20%) will be determined by the logical organization of classes in this hierarchy and by how effectively you have been able to use inheritance to minimize duplicated code. As part of this effort you must include a brief statement, as part of the javadoc at the top of your **`MaxiYatzy`** class, explaining your design decisions. See documentation and style.

Configurations

(Note that the class **`GameConfiguration`** discussed in this section is already implemented and you should not modify it.)

The design allows for many similar games to be implemented using the same basic infrastructure. You have seen above that while Yahtzee is normally played with five, six-sided dice, Maxi Yatzy is played with six. And there is no reason we couldn't use fourteen, 10-sided dice, so we make the number of dice and the maximum die value configurable. The traditional game rules also provide a 35 point bonus if the total score for the "upper section" is greater than or equal to a cutoff value of 63. Therefore, we make these two values configurable as well. We could also configure the number of rolls a player gets per turn, and we could also decide whether to all unused extra rolls to be saved and used for future turns, as in Maxi Yatzy.

All of these choices are stored in a **`GameConfiguration`** object, and you can see from the javadoc that these six constants are set in its constructor.

The MaxiYatzy class

The **`MaxiYatzy`** class manages the game state, which includes

- a **`GameConfiguration`**
- a list of **`ScoreBox`** objects forming the "upper section"
- a list of **`ScoreBox`** objects forming the "lower section"
- an instance of **`Random`** for simulating dice rolls
- a **`Combination`** representing the current turn
- the number of rolls remaining for the current turn

(There is no difference between the "upper section" and "lower section" other than that traditionally, there is a bonus for scoring above a specified cutoff value for the upper section.)

A client interacts with the game by invoking the **`startTurn()`** method, which instantiates a new **`Combination`** in which all dice are initially available. The client can observe the current state of the combination and can choose dice to "keep", i.e., move from available to complete. To roll the available dice, the client invokes the **`roll()`** method of **`MaxiYatzy`**. A client can use the **`isSatisfiedBy()`** and the **`getPotentialScore()`** methods of each score box to check whether the current combination satisfies the score box and what score would be earned if the combination was used to fill that score box.

At any point before running out of rolls, the client can choose to keep all remaining dice; otherwise, when there are no more rolls available (based on the configuration), the game should automatically move all available dice to completed. The client would then choose a score box to fill using the completed combination.

This process can be seen in action by reading the code for the sample text-based UI.

About the UI

There is a sample user interface **`TextUIMain`** provided in the default (top-level) package of the sample code. *This code will not run* until you have implemented the **`MaxiYatzi`** and **`Combination`** classes, and you will not be able to play a game until you have started implementing the necessary **`ScoreBox`** types. As you implement and test the score boxes, you can uncomment the corresponding constructors (and add appropriate import statements) in the **`ExampleGameFactory`**.

This is a text-based UI using clunky console I/O. It is not as slick as a graphical UI, but has the advantage that the code is entirely comprehensible, so you can read it to see how the other classes in the application are used. A typical screenshot of a game in progress is shown below.

```
Potential scores for this roll:
 0)    1 Ones
 1)    6 Twos
 2)    0 Threes
 3)    0 Fours
 4)    5 Fives
 5)    6 Sixes
 6)    4 One Pair
 7)    0 Two Pairs
 8)    6 Three of a kind
 9)    0 Four of a kind
10)    0 Five of a kind
11)    0 Full House
12)    0 Castle
13)    --- Tower          26 (1 1 6 6 6 6)
14)    0 Short Straight
15)    --- Full Straight 40 (1 2 3 4 5 6)
16)    0 Maxi Yatzy
17)    18 Chance
                         -----
       Upper section total: 0 (0 plus bonus 0)
       Lower section total: 66
                     SCORE: 66
You rolled 1 2 2 2 5 6 ()
a) keep all remaining
b) reroll all remaining
c) select dice to keep
Your choice: c
Enter dice to keep (separated by spaces): 2 2 2
You now have 1 5 6 (2 2 2)
You have 4 rolls left.
Press ENTER to roll the dice...
```

The dashed lines in the second column indicate the categories called "Tower" and "Full Straight" are already filled, and the rightmost columns show the score and the list of dice for the combination that was used to fill the score box. The “You rolled…” shows the current state of our dice. We evidently chose to "keep" the three 2's from the previous roll. The second column in the list shows us what potential score we would get in each score box for this roll as it stands (e.g. 6 points if we use it to fill the "Twos" category, or 18 points if we use it for the "Chance" category).

You can edit the UI main method to 1) choose a seed for the random number generator, in case you want the dice results to be reproducible while you are developing the code, and 2) to get a different game from the **`ExampleGameFactory`**. To try a different game from those constructed in the **`ExampleGameFactory`**, add another static method to **`ExampleGameFactory`**. (The "factory" keeps the UI from having any direct dependence on any of the specific category classes, so it will continue to work correctly even if you add new or different categories.)

Testing and the SpecChecker

As always, you should try to work incrementally and write tests for your code as you develop it.

Do not rely on the UI demo code for testing! Trying to test your code using a UI is very slow, unreliable, and generally frustrating. ***In particular, when we grade your work we are NOT going to run the UI, we are going to test that each method works according to its specification.***

We will provide a basic SpecChecker, but **it will not perform any functional tests of your code**. At this point in the course, you are expected to be able to read the specifications, ask questions when things require clarification, and write your own unit tests.

Since the test code is not a required part of this assignment and does not need to be turned in, **you are welcome to post your test code on Piazza for others to check, use and discuss.**

The SpecChecker will verify the class names and packages, the public method names and return types, and the types of the parameters. If your class structure conforms to the spec, you should see a message similar to this in the console output:

```
x out of x tests pass.
```

This SpecChecker will also offer to create a zip file for you that will package up everything in your hw4 package. **Please check this carefully. In this assignment you may be including one or more abstract classes of your own, in addition to the twelve required classes, so make sure they have been included in the zip file.**

Importing the sample code

The sample code is distributed as a complete Eclipse project that you can import.

1. Download the zip file **`hw4_skeleton.zip`**. You don't need to unzip it.
2. In Eclipse, go to File -> Import -> General -> Existing Projects into Workspace, click Next.
3. Click the radio button for "Select archive file".
4. Browse to the zip file you downloaded and click Finish.

If for some reason you have problems with the process above, or if the project does not build correctly, you can construct the project manually as follows:

1. Extract the zip file containing the sample code (*Note: on Windows, you have to right click and select "Extract All" – it will not work to just double-click it*)
2. In Windows Explorer or Finder, browse to the src directory of the zip file contents
3. Create a new empty project in Eclipse
4. In the Eclipse Package Explorer, navigate to the src folder of the new project.
5. Drag the **`hw4`** and **`api`** folders from Explorer/Finder into the **`src`** folder in Eclipse.

Getting started

Sometimes the best way to start on an inheritance hierarchy is, initially, to forget about inheritance. That is, if you have two classes A and B that might be related, just implement both of them completely from scratch. Then look at what they have in common. Does it make logical sense for one to extend the other? Should the common code be factored into a common, abstract superclass? **Be sure you have done and understood the second checkpoint of Lab 8 before you start making decisions about inheritance.**

1. General notes

- You can work on **`Combination`** independently from the other classes
- You can work on the technical details of the score boxes without having **`Combination`** done, since the **`isSatisfiedBy`** and **`getPotentialScore`** methods only require an int array as argument
- You can work on most of **`MaxiYatzy`** without having **`Combination`** working

2. The **`Combination`** class is something that could have been a good problem for mini2 - it's just a matter of some array and list manipulation. As always, start with some simple usage examples:

```
Combination c = new Combination(new int[] {1, 3, 2, 5, 2});
System.out.println(Arrays.toString(c.getAvailableDice()));
System.out.println("Expected [1, 2, 2, 3, 5]");
System.out.println(Arrays.toString(c.getCompletedDice()));
System.out.println("Expected []");
System.out.println();
```

(Notice that the **`getXXX`** methods always return an int array with values in ascending order. You can use Arrays.sort or Collections.sort for this.)

2. Initially all dice are "available", but we can move them to "completed" with the **`choose()`** method, like this:

```
c.choose(3);
c.choose(2);
System.out.println(Arrays.toString(c.getAvailableDice()));
System.out.println("Expected [1, 2, 5]");
System.out.println(Arrays.toString(c.getCompletedDice()));
System.out.println("Expected [2, 3]");
System.out.println(Arrays.toString(c.getAll()));
System.out.println("Expected [1, 2, 2, 3, 5]");
System.out.println();
```

How you keep track of which dice are available is up to you - you could keep two different arrays, two lists, or just use a parallel array of booleans to mark which are still available.

3. The **`Combination`** should not be "complete" when there are still available dice:

```
System.out.println(c.isComplete());
System.out.println("Expected false");
System.out.println();
```

4. We also need to be able to re-roll the available dice. Note that the randomization that produces the new die values happens elsewhere - in the **`Combination`** class, the available dice are just replaced with the given values.

```
c.updateAvailableDice(new int[] {6, 4, 2});
System.out.println(Arrays.toString(c.getAvailableDice()));
System.out.println("Expected [2, 4, 6]");
System.out.println(Arrays.toString(c.getCompletedDice()));
```

```
System.out.println("Expected [2, 3]");
System.out.println(Arrays.toString(c.getAll()));
System.out.println("Expected [2, 2, 3, 4, 6]");
System.out.println();
```

5. Once all dice are completed, **`getAvailableDice()`** returns an empty array

```
c.chooseAll();
System.out.println(Arrays.toString(c.getAvailableDice()));
System.out.println("Expected []");
System.out.println(Arrays.toString(c.getCompletedDice()));
System.out.println("Expected [2, 2, 3, 4, 6]");
System.out.println(Arrays.toString(c.getAll()));
System.out.println("Expected [2, 2, 3, 4, 6]");
System.out.println();

// should be complete now
System.out.println(c.isComplete());
System.out.println("Expected true");
System.out.println();
```

6. Next you might try a score box. **`MatchTargetScoreBox`** is relatively simple. As a first attempt, forget about inheritance: just add the declaration **`implements ScoreBox`** and stub in the missing methods so there are no compile errors. The **`isSatisfiedBy`** method always returns true and **`getPotentialScore`** just adds up the values that match the specified target:

```
ScoreBox sb = new MatchTargetScoreBox("Threes", 3);
int[] test = {1, 3, 3, 3, 3, 5, 6};
System.out.println(sb.isSatisfiedBy(test));
System.out.println("Expected true");
System.out.println(sb.getPotentialScore(test));
System.out.println("Expected 12");
```

7. If you have **`Combination`** more or less implemented, the rest of the **`ScoreBox`** methods are easy:

```
System.out.println(sb.isFilled());
System.out.println("Expected false");
System.out.println(sb.getScore());
System.out.println("Expected 0");

// try filling the score box with a completed Combination
c = new Combination(test);
c.chooseAll();
sb.fill(c);
```

```
System.out.println(sb.isFilled());
System.out.println("Expected true");
System.out.println(sb.getScore());
System.out.println("Expected 12");
Combination saved = sb.getDice();
System.out.println(c == saved);
System.out.println("Expected true");
```

Then try doing another easy one, say, **`ChanceScoreBox`**.

8. The **`MaxiYatzy`** class essentially has three responsibilities: maintain two lists of **`ScoreBox`** objects, keep track of the **`Combination`** representing the player's current turn, and count rolls. Take a look at some sample usage of the two lists first. You can do this without having **`Combination`** implemented. Note that you always need a **`Configuration`** object to construct a game:

```
GameConfiguration config = new GameConfiguration(
    4,  // four dice
    5,  // max value 5
    4,  // 4 rolls per turn
    0,  // upper section bonus
    0,   // upper section bonus cutoff
    true);  // extra rolls are saved for future turns

MaxiYatzy game = new MaxiYatzy(config, new Random(42));
game.addLowerSectionScoreBox(new MatchTargetScoreBox("Threes", 3));
game.addLowerSectionScoreBox(new ChanceScoreBox("Chance"));
ArrayList<ScoreBox> lower = game.getLowerSection();
for (ScoreBox b : lower)
{
  System.out.println(b.getDisplayName());
}
System.out.println("Expected 'Threes', 'Chance' ");
```

9. It is the **`startTurn()`** method that constructs a **`Combination`** to represent the current state of the player's dice. Initially (that is, before calling **`rollAvailableDice()`**) they all have value zero. Note that the length of the array of dice, and the possible values of the dice, are determined by the **`Configuration`**.

```
game.startTurn();
System.out.println(game.getRemainingRolls());
System.out.println("Expected 4");
Combination comb = game.getCurrentDice();
System.out.println(Arrays.toString(comb.getAvailableDice()));
System.out.println("Expected [0, 0, 0, 0]");
```

```
    game.rollAvailableDice();
    System.out.println(Arrays.toString(comb.getAvailableDice()));
    System.out.println("(Expected four random nonzero values in range 1-5)");
    System.out.println(game.getRemainingRolls());
    System.out.println("Expected 3");
```

10. We can complete the current combination and use it to fill the Chance score box. Once the score box is filled, it should become part of the total score for the game.
Since the last argument to the Configuration constructor is true, the remaining rolls are saved when the next turn is started.

```
    comb.chooseAll();
    lower.get(1).fill(comb);
    System.out.println(game.getLowerSectionTotal());
    System.out.println(game.getTotalScore());
    System.out.println("(Expected: sum of dice above used to fill Chance
box)");

    // start new turn
    game.startTurn();
    System.out.println(game.getRemainingRolls());
    System.out.println("Expected 7");
```

11. What's left is to start working out of the ScoreBox implementations. One part of the challenge is just the particular details of each concrete class. You might start by think about test cases for the **`isSatisfiedBy`** and **`getPotentialScore`** methods of some of the concrete types. For example,

```
    sb = new FullHouseScoreBox("Full House");
    System.out.println(sb.getPotentialScore(new int[] {1, 2, 3, 4, 5, 5}));
    System.out.println("Expected 0");
    System.out.println(sb.getPotentialScore(new int[] {2, 2, 3, 4, 5, 5}));
    System.out.println("Expected 0");
    System.out.println(sb.getPotentialScore(new int[] {2, 2, 3, 3, 3, 5}));
    System.out.println("Expected 18");

    sb = new NOfAKindScoreBox("3 of a kind", 3);
    System.out.println(sb.getPotentialScore(new int[] {2, 2, 2, 3, 3, 3, 3,
4}));
    System.out.println("Expected 9");
```

12. As you start implementing the concrete **`ScoreBox`** types, you'll notice similarities and common code. Think about when and whether it makes sense to move common code into an abstract superclass that several types can extend.

Requirements and guidelines regarding inheritance

A portion of your grade (15-20%) will be based on how well you have used inheritance, and abstract classes, to create a clean design with a minimum of duplicated code. Please note that there is no one, absolute, correct answer for where every method and piece of data belongs – you have design choices to make.

Specific requirements

> **You are not allowed to use non-private variables**. Call the superclass constructor to initialize its attributes, and use superclass accessor methods to access them. If your subclass needs some kind of access that isn't already provided by public methods, you are allowed to define your own **`protected`** methods or constructors.

Other general guidelines

- You should not use **`instanceof`** or **`getClass()`** in your code, or do any other type of runtime type-checking, to implement correct behavior. Rely on polymorphism.

- No class should contain extra attributes or methods that it doesn't need to satisfy its own specification

- Do not ever write code like this:

  ```
  public void foo()
  {
    super.foo()
  }
  ```

  There is almost never any reason to declare and implement a method that is already implemented in the superclass, unless you need to override it to change its behavior. *That is the whole point of inheritance!*

Style and documentation

Special documentation requirement: write a few sentences, as part of the class Javadoc for **`MaxiYatzy`**, explaining your design choices for how you organized the class hierarchy.

Roughly 10 to 15% of the points will be for documentation and code style.

Some restrictions and guidelines for using inheritance are described above.

When you are overriding a superclass or interface method, **it is usually NOT necessary to re-write the Javadoc**, unless you are really, really changing its behavior. Just include the @Override annotation. (The Javadoc tool automatically copies the superclass Javadoc where needed.)

The other general guidelines are the same as in homework 3. Remember the following:

- You must add an @author tag with your name to the class javadoc at the top of each of the classes you write or modify.
- You must javadoc each instance variable and helper method that you add, as well as specified methods that are not already documented in a supertype. Anything you add that is not in the public API must be **`private`** or **`protected`**.
- Keep your formatting clean and consistent.
- Avoid redundant instance variables
- Accessor methods should not modify instance variables

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder **`hw4`**. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag **`hw4`**. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in, and **for this assignment you are welcome to post and discuss test code**. (In the Piazza editor, use the button labeled "code" to have the editor keep your code formatting. You can also use "pre" for short code snippets.)

If you have a question that absolutely cannot be asked without showing part of your source code, change the visibility of the post to "private" so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form "read all my code and tell me what's wrong with it" will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any announcements from the that are labeled "Official Clarification" are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be placed in the

Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

What to turn in

Note: You will need to complete the "Academic Dishonesty policy questionnaire," found on the Assignments page on Canvas, before the submission link will be visible to you.

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_hw4.zip`**. and it will be located in whatever directory you selected when you ran the SpecChecker. It should contain one directory, **`hw4`**, which in turn contains the required classes, *and any others you defined in the* ***`hw4`*** *package*. **Please LOOK at the file you upload and make sure it is the right one and contains everything needed!**

Submit the zip file to Canvas using the Assignment 4 submission link and verify that your submission was successful. If you are not sure how to do this, see the document "Assignment Submission HOWTO", which can be found linked on our Canvas home page.

> We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **hw4**, which in turn should contain everything in your hw4 directory. You can accomplish this by zipping up the **hw4** directory of your project. **Do not zip up the entire project**. The file must be a zip file, so be sure you are using the Windows or Mac zip utility, and NOT a third-party installation of WinRAR, 7-zip, or Winzip.