Project 4 - Yahtzee (Object-Oriented Programming)

CS 0401 — Intermediate Programming with Java

University of Pittsburgh

Yahtzee (see `http://en.wikipedia.org/wiki/Yahtzee` for a rule overview) is a dice game in which players compete to achieve the highest possible total score. Each turn, a player rolls five dice: he/she may elect to re-roll some (or all) of the dice, or to keep the roll. A player may re-roll at most twice in any turn. Having rolled the dice, the player then chooses how best to score the roll: there are several categories (e.g., *Fives*, *Three of a Kind*, *Small Straight*), each with a different set of requirements for scoring. Over the course of the game, the player may use each category only once.

For this assignment, you will implement a number classes. **Some classes may have to implement a certain interface or to extend from another class. So, read the instruction carefully**. The final program will simulate the essential aspects of on e-player **Yahtzee**. When the code is completed, a user should be able play a game of **Yahtzee**. In other words, play exactly 13 turns (there are exactly 13 scoring catagories), and each turn a play is allowed to re-roll twice.

For this project, we will provide some tester classes. However, since most of your class must either directly implement interfaces or indirectly extend from other classes, all of your classes must have all required capabilities. Because of this aspect of OOP, if we use our own program that utilize those capabilities, it should work without any problem. **In your main program (`Yahtzee.java`), you should not call any new methods other than those specifies in interfaces. If you do, our program may not work properly with your classes**.

Part I: The Interface `DieInterface` and Class `Die` (20 Points)

Yahtzee consists of five dice. So, the first class that we are going to implement is the class `Die` where an instance of this class (object) can be used as a die. To ensure that all your dice will have the required capacities, your class `Die` must implement the `DieInterface`. The following is the content of `DieInterface.java`:

```
public interface DieInterface
{
    public static String[] dieFaces =
        {"+---+\n|   |\n| o |\n|   |\n+---+",
         "+---+\n|o  |\n|   |\n|  o|\n+---+",
         "+---+\n|o  |\n| o |\n|  o|\n+---+",
         "+---+\n|o o|\n|   |\n|o o|\n+---+",
         "+---+\n|o o|\n| o |\n|o o|\n+---+",
         "+---+\n|o o|\n|o o|\n|o o|\n+---+"};
```

```
    public static String toDieString(DieInterface aDie)
    {
        return dieFaces[aDie.getFaceValue() - 1];
    }

    // Do not modify above this line

    public static String toDiceString(DieInterface[] dice)
    {
        StringBuilder result = new StringBuilder();

        // Add your code here

        return result.toString();
    }

    // Do not modify below this line
    public int roll();
    public int getFaceValue();
}
```

Your class `Die` must implement the above `DieInterface`. In doing so, Java will enforce you to implement the following capabilities (the last two methods above):

- `int roll()`: This method allows us to roll a die and get its value (between 1 and 6)
- `int getFaceValue()`: This method allows us to look at a die face that has been rolled and see what is its current value.

The header of your `Die` class must look like the following:

```
public class Die implements DieInterface
{
   :
}
```

Note that there are some useful methods related to dice such as generating a string that looks like a die face or generating a string representing a row of die faces with die number underneath each dice. For this two useful methods, you must implement them as static methods (class method). Look at the above code of `DieInterface`, you will see how the `toDieString()` method was implemented. This method generates a string that looks like a die face when printed. Your job is to complete the `toDiceString()` method that take an array of dice and generate a string representation of a row of dice. The following is an example of output strings of methods `toDieString()` and `toDiceString()`:

```
+---+
|o o|
|   |
|o o|
```

```
+---+
+---+  +---+  +---+  +---+  +---+
|o o|  |   |  |o o|  |o  |  |o o|
|o o|  | o |  |   |  | o |  |   |
|o o|  |   |  |o o|  |  o|  |o o|
+---+  +---+  +---+  +---+  +---+
  1      2      3      4      5
```

The above output can be generated using the following code snippet:

```
DieInterface aDie = new Die();
System.out.println(DieInterface.toDieString(aDie));

DieInterface[] dice = new DieInterface[5];
for(int i = 0; i < 5; i++)
    dice[i] = new Die();
System.out.println(DieInterface.toDiceString(dice));
```

Notice how the above code snippet utilizes interface as a reference type.

Once you complete the `DieInterface` and your class `Die` is implemented, you should test your class to ensure that it works properly. The class `DieTester.java` and an example of its output `DieTesterOutput.txt` are given. Make sure to read the code together with the output to understand the effect of each method.

IMPORTANT: You are not allowed to add a new method or method declarations in the file `DieInterface.java`

Part II: The Class `Score` (xx Points)

In Yahtzee, there are the total of thirteen categories of scoring. Each score has its name and its number of points (in integer). The number of points for each scoring is based on face values of all 5 dice. In a game, you can use each type of scoring only once. When the game is over, the score of the player is the sum of all thirteen scores.

From the above observation, we can either create 13 classes (one for each scoring type) or we can use inheritance by creating a common superclass which contains all capability and those similarities among scoring categories. Then each subclass can simply extend this superclass with a very minimum implementation inside it.

To ensure that each scoring category will have the right capabilities, we are again going to use the interface named `ScoreInterface` (`ScoreInterface.java`) as shown below:

```
public interface ScoreInterface
{
    public String getName();
    public int getDiceScore(DieInterface[] dice);
    public int getScore();
    public void setScore(DieInterface[] dice);
    public boolean isUsed();
```

```
    public void reset();
}
```

From the above interface, a class that implements `ScoreInterface` must have the following capabilities:

- `String getName()` returns the string representation of the name of this score. The string representation will be set by a user which should be defined by the user when he/she construct an object.
- `int getDiceScore(DieInterface[] dice)` returns the integer value representing the score of this category based on the given array of five dice. Note that this is like a score calculator for this category. **By calling this method, it does not set that this scoring category has been used**. It will be used to show to the user how many points they will get if they use this category.
- `int getScore()` returns the number of points that a player has for this category. If it has not been used yet, this method should return 0.
- `void setScore(DieInterface[] dice)` sets that this category has been used and the number of points is based on the array of five dice. **Obviously, this method should use the `getDiceScore()` method to calculate the number of points before it set its score**.
- `boolean isUsed()` returns true or false based on whether this category has been used.
- `void reset()` resets the score of this category back to 0 and set that this category has not been used yet.

If you look closely to the above capabilities, you may observe that all thirteen cateories of scoring type are pretty much the same in the sense that:

- They all have names and can be read by the `getName()` method
- They can be set by the `setScore()` method
- Their scores can be read by the `getScore()` method
- Whether they have been used can be checked by the `isUsed()` method
- They can be reset of the `reset()` method

The only different is how each category calculate its number of points based on all five dice.

Because of the above observation, you should implement the class `Score` which will be a superclass of all scoring classes. You class `Score` must be declared **abstract** because it is going to implement all capabilities that are similar among categories and leave out the one that is different, the `getDiceScore()` method. This will make sure that a class that extends the class `Score` will have to implement the `getDiceScore()` method.

Note that the header of your class `Score` should look like the following:

```
public abstract class Score implements ScoreInterface
{
    :
}
```

All similar capabilities (methods) should be implemented in this class. Again, when a class extends from this class, that class will only need to implement exactly one method, the `getDiceScore()` method because all other capabilities has been implemented via inheritance.

For the class `Score`, use the following `toString()` method:

```
public String toString()
{
    String scoreString = String.format("%3d", score);
    return name + ": " + scoreString;
}
```

where the variable `name` is an instance variable of type `String` which is the user defined string that a user used when constructing this object. The variable `score` is an instant variable of type `int` that stores the number of points of this category (0 if it has not been set yet).

IMPORTANT: You are not allowed to add a new method or method declarations in the file `ScoreInterface.java`

Part III: Scoring Classes

For this project, you will have one object associated with one category of scoring. In other word, you will have the total of 13 objects. These objects will be used as score calculators as well as displaying a Yahtzee score card.

The scoring of **Yahtzee** consists of two main sections, the upper section and the lower section. We will look into each section separately.

The Upper Section

The upper section consists of six categories, Ones, Twos, Threes, Fours, Fives, and Sixes. For this scoring section, you simply total only the specified die face. For example, if you roll 5, 3, 1, 3, 1, the score of each category will be as follows:

- **Ones** is 2 since there are two dice with value 1 (1 + 1 = 2).
- **Twos** is 0 since there are no dice with value 2.
- **Threes** is 6 since there are two dice with value 3 (3 + 3 = 6).
- **Fours** is 0.
- **Fives** is 5 since there is one die with value 5.
- **Sixes** is 0.

As you may notice, the algorithm how to calculate the number of points for each of these categories are pretty much the same. Because of that, you will create a single class named `UpperSectionScore` which will be used to construct the total of six objects (one for each category). **The `UpperSectionScore` class must extend the `Score` class to ensure that it has all capabilities defined in the `ScoreInterface`.** Your job is to implement an appropriate constructor and the `getDiceScore()` method (the one that you did not implement in the `Score` class). The header of the class `UpperSectionScore` must be

```
public class UpperSectionScore extends Score
{
    :
}
```

The test program (`UpperSectionScoreTester.java`) is given. Let's look at its code a little bit:

```
public class UpperSectionScoreTester
{
    public static void main(String[] args)
    {
        String[] names =
           {"  Ones", "  Twos", "Threes", " Fours", " Fives", " Sixes"};
        ScoreInterface[] scores = new ScoreInterface[6];
        for(int i = 0; i < 6; i++)
            scores[i] = new UpperSectionScore(names[i], i + 1);

        DieInterface[] dice = new DieInterface[5];
        for(int i = 0; i < 5; i++)
            dice[i] = new Die();

        System.out.println(DieInterface.toDiceString(dice));

        for(int j = 0; j < 6; j++)
            System.out.println(scores[j].getName() + ": " +
                               scores[j].getDiceScore(dice));
    }
}
```

Note that the above program uses the `ScoreInterface` as a reference type. The constructor of the class `UpperSectionScore` takes exactly two arguments and it has the following signature:

```
public UpperSectionScore(String aName, int aNumber)
```

The first parameter is for a user-defined name (of type `String`) and a number (of type `int`). The given number (`aNumber`) is used to determine how to calculate the number of points. For the category Ones, the give number will be 1, for the category Twos, the given number will be 2, and so on (see the code above). **Make sure to use this number to adjust your algorithm. Do not use name**. Here is an example of a result from the above program:

```
+---+  +---+  +---+  +---+  +---+
|o o|  |o o|  |o  |  |o o|  |o o|
|   |  |o o|  | o |  |   |  | o |
|o o|  |o o|  |  o|  |o o|  |o o|
+---+  +---+  +---+  +---+  +---+
  1      2      3      4      5
  Ones: 0
```

```
  Twos: 0
Threes: 3
 Fours: 8
 Fives: 5
 Sixes: 6
```

Again, note that all references related to this project are **interfaces**, `ScoreInterface` and `DieInterface`.

Lower Section

The lower section consists of seven categories. The scoring of each category are as follows:

- **Three of a Kind**: If a roll contains at least three of the same die faces, simply total all the die faces and that is the score of this category.
- **Four of a Kind**: The scoring of this category is the same as the **Three of a Kind** (total all the die faces) except that you must have at least four of the same die faces. Obviously, if you get four of a kind, you also get three of a kind.
- **Full House**: If you have three of a kind and a pair (e.g., 5, 3, 5, 5, 3), the score of this category is 25.
- **Small Straight**: If you have a sequence of 4 consecutive die faces (e.g., 2, 4, 3, 4, 5 which contains 2, 3, 4, 5), 30 points.
- **Large Straight**: If you have a sequence of 5 consecutive die faces, 40 points. Again, if a player get large straight, he/she also get small straight.
- **Chance**: For this category, simply total all the die face values.
- **Yahtzee**: This category is simply 5 of a kind. This score of this category is 50 points.

For this lower section, it can be separated into four main sub-section:

- **Of A Kind**: for three of a kind, four of a kind, and five of a kind (Yahtzee)
- **Full House**:
- **Straight**: for small straight and large straight
- **Chance**:

So, for the lower section, you need four more classes (one for each sub-section) as follows:

- `OfAKind`: Three objects of this class should be constructed one for three of a kind, one for four of a kind, and one for five of a kind (Yahtzee). The constructor of this class should have the following signature:

  ```
  public OfAKind(String aName, int numSameFaces)
  ```

  where `numSameFaces` indicates how many die faces that a player should have to satisfy this category as shown below:

```
ScoreInterface s1 = new OfAKind("Three of a Kind", 3);
ScoreInterface s2 = new OfAKind(" Four of a Kind", 4);
ScoreInterface s3 = new OfAKind("        Yahtzee", 5);
```

- `FullHouse`: One object of this class should be constructed to handle the full house category. The constructor of this class should simply take a user-define name (as a `String`) as a parameter as shown below:

```
public FullHouse(String aName)
```

- `Straight`: Two objects of this class should be constructed, one for small straight, and another for large straight. The constructor of this class should have the following signature:

```
public Straight(String aName, int numConsecutiveFaces)
```

  where `numConsecutiveFaces` indicates how many consecutive faces that a player should have to satisfy this scoring category as shown below:

```
ScoreInterface s1 = new Straight(" Small Straight", 4);
ScoreInterface s2 = new Straight(" Large Straight", 5);
```

- `Chance`: One object of this class should be constructed to handle the category chance. The constructor should simply take a string for its name.

The test program (`LowerSectionScoreTester.java`) is given and shown below:

```
public class LowerSectionScoreTester
{
    public static void main(String[] args)
    {
        ScoreInterface[] scores = new ScoreInterface[7];
        scores[0] = new OfAKind("Three of a Kind", 3);
        scores[1] = new OfAKind(" Four of a Kind", 4);
        scores[2] = new FullHouse("     Full House");
        scores[3] = new Straight(" Small Straight", 4);
        scores[4] = new Straight(" Large Straight", 5);
        scores[5] = new Chance("         Chance");
        scores[6] = new OfAKind("        Yahtzee", 5);

        DieInterface[] dice = new DieInterface[5];
        for(int i = 0; i < 5; i++)
            dice[i] = new Die();

        System.out.println(DieInterface.toDiceString(dice));

        for(int j = 0; j < 7; j++)
            System.out.println(scores[j].getName() + ": " +
```

```
                                scores[j].getDiceScore(dice));
    }
}
```

Note that the above program uses the `ScoreInterface` as a reference type. Here is an example of a result from the above program:

```
+---+  +---+  +---+  +---+  +---+
|o o|  |o o|  |o o|  |o o|  |o o|
|o o|  |   |  |   |  |   |  |   |
|o o|  |o o|  |o o|  |o o|  |o o|
+---+  +---+  +---+  +---+  +---+
  1      2      3      4      5
Three of a Kind: 22
 Four of a Kind: 22
     Full House: 0
 Small Straight: 0
 Large Straight: 0
         Chance: 22
        Yahtzee: 0
```

Grading Rubric

For his section, the number of point of each class is as follows:

- `UpperSectionScore`: xx points
- `OfAKind`: xx points
- `FullHouse`: xx points
- `Straight`: xx points
- `Chance`: xx points

Part IV: The Class `Yahtzee` (xx Points)

The class `Yahtzee` will be the main program that simulate one-player Yahtzee. Your program has to perform the following tasks:

1. As the player whether he/she wants to play Yahtzee
2. If the answer is `y` (for yes), as the player to enter his/her name. If the answer is `n` (for no), simply terminate the program.
3. Show the current Yahtzee score card
4. Roll all five dice
5. Ask the player to enter the die number he/she wants to keep

6. Roll all dice that the player does not want to keep
7. Ask the player to enter the die number he/she wants to keep
8. Roll all dice that the player does not want to keep
9. Show all **unused** score categories and ask the player which one he/she wants to use
10. Go back to step 3 if there are unused score categories. Otherwise, go to the next step
11. Show the final score card
12. Go back to step 1

An example of a play is shown in `YahtzeeOutput.txt`. Make sure to look at the file to see how a score card should be displayed and input format for selecting which dice to keep. Also note that once a scoring category has been used, it will not be shown in step 9.

Constraints

Make sure you follow the following constraints:

1. In your `Yahtzee.java`, it must not contain the following reference type; `Die`, `Score`, `UpperSectionScore`, `OfAKind`, `FullHouse`, `Straight`, and `Chance`. Any reference related to dice and scoring must be `DieInterface` and `ScoreInterface` only.
2. You are not allowed to modify `DieInterface.java` other the inserting the code snippet for the static method `toDiceString()`
3. You are not allowed to modify `ScoreInterface.java`
4. You can add more methods as you see fit. But remember that if those methods are added to any of those classes, you will not be able to call them from the class `Yahtzee` because they are but declared in either `DieInterface` or `ScoreInterface`. Again, you are not allowed to add any more method declarations into both `DieInterface` and `ScoreInterface`. Thus, those added methods will be helps methods for your classes but not for `Yahtzee` class.
5. You can use the class `ArrayList` if you want to.
6. For this project, we will assume that the player will always enter valid inputs

Submission

The due date of this project is stated on the CourseWeb. Late submissions will not be accepted. For this project, there are multiple Java file that you have to submit. **Make sure to zip them into a single file named `project4.zip`. For this project, you have to submit the following files:**

- `DieInterface.java`
- `Die.java`
- `Score.java`

- `UpperSectionScore.java`
- `OfAKind.java`
- `FullHouse.java`
- `Straight.java`
- `Chance.java`
- `Yahtzee.java`

Zip them in to one file named `project4.zip` and submit the file `project4.zip` via CourseWeb. No late submission will be accepted.