COMP1721 Object-Oriented Programming

Coursework 2

1 Introduction

Your task is to implement a simulation of the card game **Baccarat**—specifically, the simpler 'punto banco' variant of the game. To assist you, we have provided three Java classes: `Card`, `CardCollection` and `CardException` These should form the basis of your solution.

Please note: an absolute requirement of this assignment is that you should not alter the definitions of the `Card`, `CardCollection` and `CardException` classes in any way.

2 Preparation

1. Start by learning the rules of the game! Consult the Wikipedia page on Baccarat, which has a very good summary. **Note that you only need to read the sections headed 'Punto banco' and 'Tableau of drawing rules'.**
2. Download the Zip archive `cwk2files.zip` from Minerva.
3. Unzip the Zip archive. You can do this from the command line in Linux, macOS and WSL 2 with `unzip cwk2files.zip`. This will give you a subdirectory named `cwk2` containing all the files you need. **Then remove the Zip archive, to ensure that you don't accidentally submit it as your solution!**
4. Study the files in `cwk2` In particular, examine the file `README.md`, as this provides guidance on how to run the tests on which your mark will be partly based.

3 Basic Solution

For the basic solution, you must implement these classes:

- `BaccaratCard`, to represent a single playing card in Baccarat
- `BaccaratHand`, to represent a hand of cards in Baccarat
- `Shoe` to represent the 'shoe' from which cards are dealt in Baccarat

You will also need to implement a small program in `Baccarat.java` that uses the classes.

The files for all of these classes can be found in `src/main/java` **Your solution MUST be implemented in these files, in this specific location!**

Figure 1 shows the relationships that should exist between the three main classes, and the methods they must support. Relationships with the other provided classes are not shown.

Before writing any code, think carefully about the best way to reuse the `Card` and `CardCollection` classes that we have provided to you.

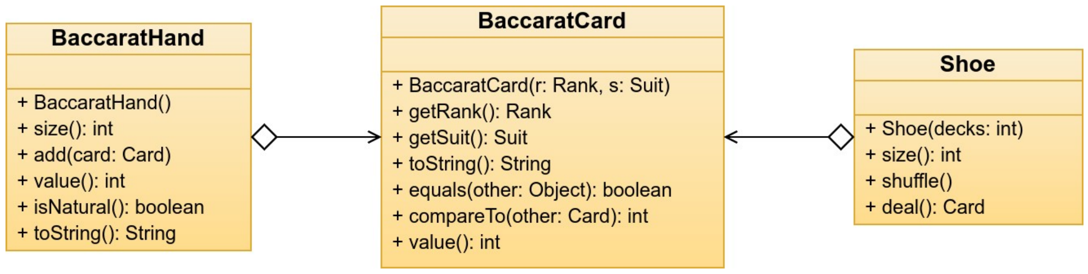

Figure 1: Classes needed for Baccarat simulation.

As in Coursework 1, we have provided tests that will help you check whether you have implemented the classes correctly. Some of the code needed to pass the tests is in the classes we have provided, but you will need to also write stubs for some of the methods in Figure 1 in order for the tests to compile and run—see the instructions for Coursework 1 if you need a reminder of how to do this.

As in Coursework 1, the tests are run using Gradle:

```
./gradlew test
```

Omit the `./` prefix if running in a Windows command shell, or use `.\gradlew.bat` to invoke Gradle if running in Windows Powershell. If using IntelliJ, you should also be able to run the tests and other coursework-related tasks via that IDE's Gradle tool window.

The following sections provide further details of the requirements for each of the three classes.

3.1 `BaccaratCard`

These are the minimum requirements for the tests to pass.

- It should be possible to create a `BaccaratCard` object by specifying a rank and a suit, where the rank is an instance of the enum `Card.Rank` and the suit is an instance of the enum `Card.Suit`
- It should be possible to query rank and suit via the `getRank()` and `getSuit()` methods.
- Calling `toString()` on a `BaccaratCard` object should return a two-character string in which the first character is the rank (A, 2, 3 . . . 9, T, J ,Q, K) and the second character is the Unicode symbol for the card's suit: ♣ (`\u2663`), q (`\u2666`), r (`\u2665`), or ♠ (`\u2660`).
- It should be possible to compare `BaccaratCard` objects in two ways, using the `equals()` and `compareTo()` methods. These methods should have the expected behaviour for Java classes, as discussed in the lectures. They should use rank and suit to perform their comparisons.
- The `value()` method of `BaccaratCard` should return the points value of the card in the game of Baccarat. (See the Wikipedia article for details of scoring.)

Remember that we have provided a `Card` class to help you with implementation. Some of the requirements above are satisfied by code in that class. Think about the best way of reusing this code.

3.2 `BaccaratHand`

These are the minimum requirements for the tests to pass.

- A `BaccaratHand` should be able to store `BaccaratCard` objects, but it should start out as empty.
- It should be possible to add a `BaccaratCard`object to a `BaccaratHand` by calling the `add()` method.
- Calling `size()` on a `BaccaratHand` should return the number of cards in the hand.
- Calling `toString()` on a `BaccaratHand` should return a string containing two-character representations of each card, separated from each other by a space. For example, a hand containing the Ace of Clubs, Four of Diamonds and Jack of Spades should yield "A♣ 4q J♠".
- The `value()` method of `BaccaratHand` should return the points value of a hand in the game of Baccarat. (See the Wikipedia article for details of scoring.)
- The `isNatural()` method of `BaccaratHand` should return `true` if the hand has two cards and a points value of 8 or 9, `false` otherwise.

Remember that we have provided a `CardCollection` class to help you with implementation. Some of the requirements listed above are satisfied by code in that class. Think about the best way of reusing this code.

3.3 `Shoe`

These are the minimum requirements for the tests to pass.

- `Shoe` must have a constructor in which the number of decks of cards is specified as a parameter. This can have values of 6 or 8; any other value should result in a `CardException` being thrown.
- The constructor of `Shoe` should ensure that a shoe stores the specified number of complete decks of `BaccaratCard` objects. A deck of cards is the full set of 52 cards, ordered first by suit and then by rank. The constructor should not reorder the cards in any way.

- Calling `size()` on a `Shoe` should return the number of cards in the shoe.
- The `shuffle()` method of `Shoe` should reorder the cards in the shoe randomly. Java's `Collections` utility class, from the `java.util` package, will help you implement this easily. See the documentation of this class for more details.
- The `deal()` method of `Shoe` should remove the first stored card and return it to the caller. Attempting to deal from an empty shoe should trigger a `CardException`

Remember that we have provided a `CardCollection` class to help you with implementation. Some of the requirements above are satisfied by code in that class. Think about the best way of reusing this code.

3.4 Main Program

The final part of the Basic solution is to implement a small program in `Baccarat.java` that uses the three classes discussed earlier. This program should

- Create a shoe containing 6 full decks of cards, and shuffle it
- Create hands for the banker and a single player
- Deal a card to player then banker, then repeat this so each hand has two cards
- Display the contents and value of each hand
- Inform the user if the player or banker has a Natural

You can run the program via Gradle, with

```
./gradlew run
```

Figure 2 gives an example of the output that the program should generate.

```
Player: 8♥ T♥ = 8
Banker: 7♣ 3♥ = 0
Player has a Natural
```

Figure 2: Sample output for the Basic solution.

4 Full Solution

For the Full solution, replace the program in `Baccarat.java` with a more complete simulation of the game. Your simulation should follow a simplified version of the 'punto banco' rules outlined in the Wikipedia article. In this version, there is no betting and you should not 'burn' any cards from the shoe.

In this more complete simulation, the program will need to play multiple rounds, indicating the result (Banker win, Player win or Tie) after each round. See the subsections below for further details.

Note: marks for the Full solution will be awarded based on how much of the required behaviour you are able to implement correctly, and also on how well designed your solution is. A fully object-oriented implementation in which different aspects of the task are broken out into separate methods will score more marks than an unstructured implementation where everything is contained within the `main` method.

4.1 Interactive Mode

If the `Baccarat` program is run with a single command line argument whose value is either `"-i"` or `"--interactive"`, then it should operate in **interactive mode**. In this mode, the program should continue playing the game provided that there are at least six cards remaining in the shoe and the user has indicated a wish to continue. The program should check this with the user at the end of each round. If the user's response does not begin with either `'y'` or `'Y'` then the game should end.

At the end of the game, the program should display counts of the number of rounds played, the number of player wins, the number of banker wins and the number of tied rounds. After displaying this information, the program should terminate.

A sample of the expected program output can be seen in Figure 3.Here, the user indicated that they wanted to finished after playing ten rounds.

```
Round 8
Player: 6♦ 7♥ = 3
Banker: 7♦ K♦ = 7
Dealing third card to player...
Player: 6♦ 7♥ Q♠ = 3
Banker: 7♦ K♦ = 7
Banker win!
Another round? (y/n): y

Round 9
Player: 7♥ 2♥ = 9
Banker: A♠ K♥ = 1
Player win!
Another round? (y/n): y

Round 10
Player: 9♠ J♣ = 9
Banker: 3♦ 6♣ = 9
Tie
Another round? (y/n): n

10 rounds played
2 player wins
6 banker wins
2 ties
```

Figure 3: Sample output for Full solution (interactive mode).

To test interactive mode using Gradle, do

```
./gradlew interact
```

4.2 Non-Interactive Mode

If the `Baccarat` program is run without command line arguments, it should operate without user interaction, playing rounds of the game repeatedly until there are fewer than 6 cards remaining in the shoe. At this point, the program should display the same ‘end of game’ statistics as it does in interactive mode. The program should then terminate.

Non-interactive mode should be the behaviour seen when running the program with

```
./gradlew run
```

5 Submission

Use Gradle to generate a Zip archive containing all the files that need to be submitted:

```
./gradlew submission
```

This produces a Zip archive named `cwk2.zip` Submit this file to Minerva, via the appropriate link in the ‘Submit My Work’ folder.

The deadline for submissions is **2 pm on 2 May 2024**. The standard university penalty of 5% of available marks per day will apply to late work, unless an extension has been arranged due to genuine extenuating circumstances.

Note that all submissions will be subject to automated plagiarism checking.

6 Marking

20	Tests for main classes (BaccaratCard, BaccaratHand , Shoe)
10	Code structure and style for main classes
5	Basic Baccarat program functionality, if Full solution not attempted
15	Full Baccarat program functionality (replaces Basic)
10	Full Baccarat program structure and style
60	

Note: there are either 5 marks or 15 marks available for correct functioning of the program that uses the classes, depending on whether you've attempted the Basic solution or the Full solution. You are also assessed on program structure and style if you attempt the Full solution, but not if you attempt the Basic solution (as the amount of code required in this case is very small).

Thus the maximum mark that can be achieved for the Basic solution is 36. You can achieve up to 60 marks if you attempt the Full solution.