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 can 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 files for the assignment from Minerva or Teams, as the Zip archive `cwk2-files.zip`. **Put this file in the `coursework` directory of your repository**.
3. Unzip the Zip archive. You can do this from the command line in Linux, macOS and WSL 2 with `unzip cwk2-files.zip`.
4. Make sure that you have these files and subdirectories immediately below `coursework/cwk2`:

   ```
   README.html      config/     gradle/               gradlew.bat
   README.md        core/       gradle.properties     settings.gradle
   build.gradle     game/       gradlew
   ```

   IMPORTANT: Make sure that this is exactly what you see! For example, you should NOT have a subdirectory of `cwk2` that is itself named `cwk2`. Thus the path to the README file, relative to the repository directory, should be `coursework/cwk2/README.md`. Fix any problems with the directory structure before proceeding any further.
5. Remove `cwk2-files.zip`. Use Git to add and commit the new files, then push your commit up to gitlab.com. The following commands, executed in a terminal window while in the `coursework` directory of your repository, will achieve all of this:

   ```
   git add cwk2
   git commit -m "Initial files for Coursework 2"
   git push
   ```

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

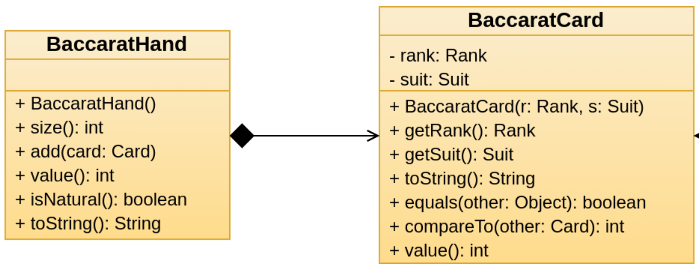
The files for these classes can be found in `core/src/main/java/comp1721/cwk2`.

As a minimum, these three classes should support the methods shown in Figure 1. **Before writing any code, think carefully about the best way to reuse the classes we have already provided.**

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 write stubs for some of the methods in Figure 1 in order for the tests to compile and run.

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

```
./gradlew :core:test
```



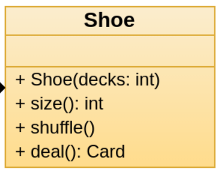


Figure 1: Classes needed for Baccarat simulation

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. `BaccaratCard` can make other methods available to users of the class besides these ones.

- It should be possible to create a `BaccaratCard` object by specifying a rank and a suit.
- Calling `toString()` on a `BaccaratCard` object should return a two-character string—e.g., `"AC"` for the Ace of Clubs, `"2D"` for the Two of Diamonds, `"TH"` for the Ten of Hearts, `"QS"` for the Queen of Spades.
- It should be possible to compare `BaccaratCard` objects in two ways, using the `equals()` and `compareTo()` methods. These methods should have the expected behaviour, as described in Lectures 6 and 9. 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. Think about the best way of reusing this code.

3.2 `BaccaratHand`

These are the minimum requirements for the tests to pass. `BaccaratHand` can make other methods available to users of the class besides these ones.

- A `BaccaratHand` should be able to store `BaccaratCard` objects, but it should start out as empty.
- It should be possible to add `BaccaratCard` objects to a `BaccaratHand` by calling `add()`.
- 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 `"AC 4D JS"`.
- 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 a points value of 8 or 9, `false` otherwise.

Remember that we have provided a `CardCollection` class to help you with implementation. Think about the best way of reusing this code.

3.3 `Shoe`

These are the minimum requirements for the tests to pass. `Shoe` can make other methods available to users of the class besides these ones.

- `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.
- 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 of the method you need to use.
- The `deal()` method of `Shoe` should remove the first stored card and return it to the caller.

Remember that we have provided a `CardCollection` class to help you with implementation. Think about the best way of reusing this code.

4 Intermediate Solution

For the intermediate solution, you must implement a class named `Baccarat`. The file for this class can be found in `game/src/main/java/comp1721/cwk2`.

The `Baccarat` class should contain a program that simulates a small part of the game of Baccarat. Your program must use all three of the classes developed in the basic solution. The program should deal two hands of cards from a shuffled shoe, one for the player and one for the banker. Each hand should contain two cards. The program should display the contents of these hands and their corresponding values. Finally, it should indicate if either hand is a 'natural'.

Here is an example of program output:

```
Player: 4D 5S = 9
Banker: 9S 3C = 2
Player has a Natural
```

You can run the program from Gradle with

```
./gradlew :game:run
```

5 Full Solution

This involves the same class as the intermediate solution. However, for the full solution, your `Baccarat` class must contain a program that plays a complete game of Baccarat, following a slightly simplified version of the 'punto banco' rules outlined in the Wikipedia article. In this simplified version, there is no betting. Also, we do not 'burn' any cards from the shoe, otherwise the rules are as shown in the article.

The program should have two modes of operation. The default mode, where no command line arguments have been supplied, is to play the game without human interaction as a series of rounds, stopping when there are less than 6 cards remaining in the shoe. The program can be run in this mode from Gradle, with

```
./gradlew :game:run
```

The other mode, interactive mode, is triggered by running with program with the `-i` option specified on the command line. You can do this via Gradle with

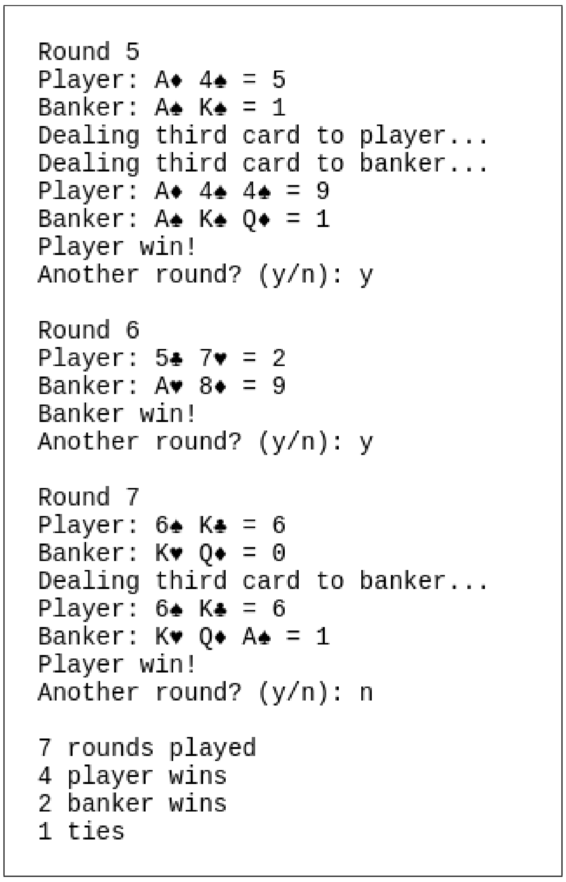
```
./gradlew :game:interactive
```

In interactive mode, the program should prompt the user after each round of the game, asking whether they wish to play another round or not—but it should do this only if there are at least 6 cards remaining in the shoe.

When play finishes, either because there aren't enough cards left in the shoe or because the user wanted to stop, 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.

A sample of program output can be seen in Figure 2. This is for interactive mode. Output should look the same in non-interactive mode, but without the prompts for user input. This example uses 'fancy' symbols for the card suits, but this isn't required; your solution can use the regular characters `C`, `D`, `H` and `S` instead, if you wish.

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



```
Round 5
Player: A♦ 4♠ = 5
Banker: A♠ K♠ = 1
Dealing third card to player...
Dealing third card to banker...
Player: A♦ 4♠ 4♠ = 9
Banker: A♠ K♠ Q♦ = 1
Player win!
Another round? (y/n): y

Round 6
Player: 5♣ 7♥ = 2
Banker: A♥ 8♦ = 9
Banker win!
Another round? (y/n): y

Round 7
Player: 6♠ K♣ = 6
Banker: K♥ Q♦ = 0
Dealing third card to banker...
Player: 6♠ K♣ = 6
Banker: K♥ Q♦ A♠ = 1
Player win!
Another round? (y/n): n

7 rounds played
4 player wins
2 banker wins
1 ties
```

Figure 2: Sample output for the full solution (interactive mode)

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.

6 Submission

There is a README file at the top level of your repository, explaining the submission process. You can open the local copy of this file to read it, or you can read the rendered version of the file on your repository's home page on gitlab.com.

Follow the instructions in the README carefully. Remember to submit `cwk2.zip` via the link provided in Minerva.

The deadline for submissions is **10 am on 4 June 2021**. 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.

7 Marking

15	Tests for basic solution
3	Code design / use of Java, basic solution
8(3)	`Baccarat` program functionality
3	Code design / use of Java, full solution
4	Coding style and comments
2	Use of version control
35	

(`Baccarat` program: 3 marks for intermediate solution, or 8 marks for full solution.)