

## Chanco!

Chanco is a card game played by 2 to 13 players that uses a standard deck of 52 playing cards. Each card has a Rank and Suit value. There are 13 Ranks (A,2,3,4,5,6,7,8,9,10,K,Q,J) and 4 Suits (Clubs:♣,Diamonds:♦,Hearts:♥,Spades:♠) making up the 52 cards in a standard-deck.

## Set-Up

The players of the game are seated in a circle. 'Card piles' that may contain up to one card are located to the left and right of each player. A Chanco game consisting of  $p$  players uses a 'game-deck' that is constructed from  $(4 \times p)$  playing cards. A game-deck is constructed such that whenever a card is added to the deck, all suits of that card's rank are also added.

In addition to the players, a dealer watches over and directs the course of the game. The dealer obtains a pre-shuffled game-deck (shuffled via a combination of cuts and riffle-shuffles) and arbitrarily deals four cards from the deck to each player.

## The Game

The objective of each player is to try and obtain a hand with all four cards having the same rank.

The game is played in rounds. At the beginning of each round, the dealer is notified if any players have won, in which case he formally congratulates them and ends the game. Assuming no one has won; each player selects a single card from their hand and discards it. The selected card is placed on the card pile to the player's left. Each player then picks up the card discarded by the opponent from the card pile on their right. The dealer then calls for another round of the game to be played. An end-game scenario is illustrated in Figure 1.

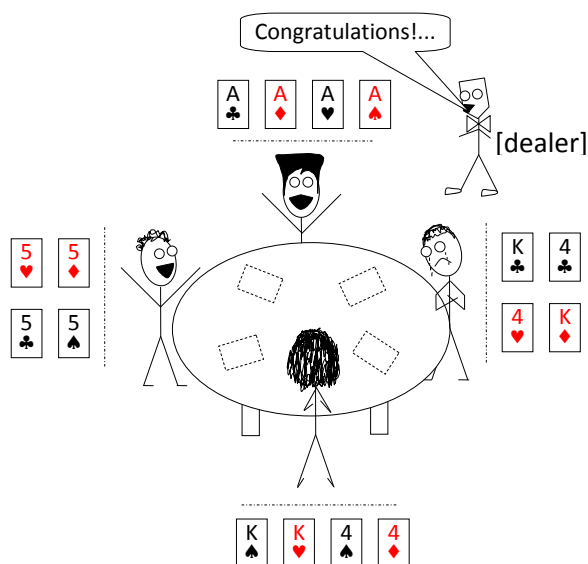


Figure 1: A Friendly Game of Chanco

## Problem Description

Your task is to complete the provided implementation of the Chanco game in Java. In addition to this specification, you should carefully read the commented source files.

You are given:

- For your convenience, a *partial* UML class-diagram describing the architecture of the game. Your implementation will build upon the given design.
- Interfaces `Player`, `IMinHeap<T extends Comparable<T>>` and `Deck` which expose the operations performed by: a player, a generic min-heap, and a deck of cards respectively.
- An abstract class `AbstractPlayer` that provides a skeletal implementation of the interface `Player`.
- Concrete classes `Card`, `ManualPlayer`, `CardPile` which model the playing card, a human player sitting at a computer terminal and a pile of cards respectively.
- A partial implementation of the `Deck` interface, `MinHeapDeck`, whose internal structure uses the min-heap abstract data type.
- The partial implementation of the class `Chanco`, that is responsible for creating the players, creating and shuffling a game-deck and instructing the dealer to begin the game.
- An enumeration, `Rank`, covering the set of rank values in a standard deck of cards.
- An enumeration, `Suit`, covering the set of suit values in a standard deck of cards.
- Some auxiliary classes `HeapException` and `HeapEntry<T>`.

You will need to write:

1. The class `MinHeap<T>` which realizes the interface `IMinHeap<T extends Comparable<T>>` and provides a generic implementation for a min-heap abstract data type.
2. Two methods: `public Deck cut()` and `public void riffleShuffle(Deck deck)` within the provided class `MinHeapDeck` which will perform some card shuffling operations.
3. The subclass `AutoPlayer` representing a computer player with a basic card selection strategy.
4. The class `Dealer`, modelling the behaviour of the game's dealer.
5. The static method `setupPlayers()` within the class `Chanco` that creates an initial setup of players as shown in in Figure 5.

### Important (Administrative) Notes:

- Refer to the test coversheet for an explanation of how to access the provided Java source files.
- All classes are members of the default (empty) package.
- Implementation of the five tasks above will require the modification of *only* the grey coloured classes shown in Figure 2.
- You may *not* introduce any additional files into the system.

## System Architecture

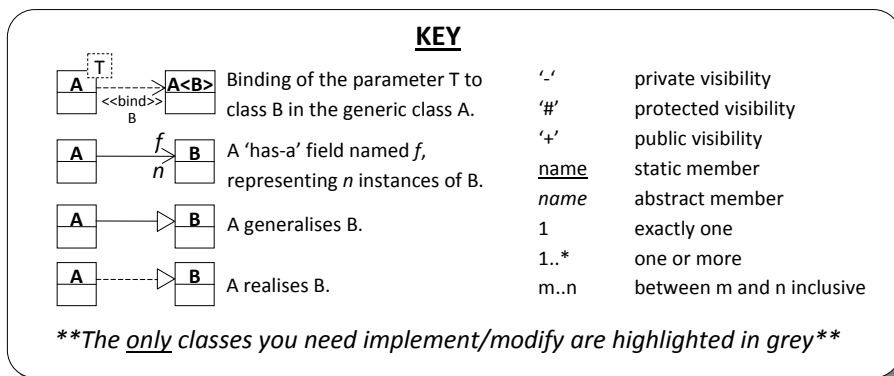
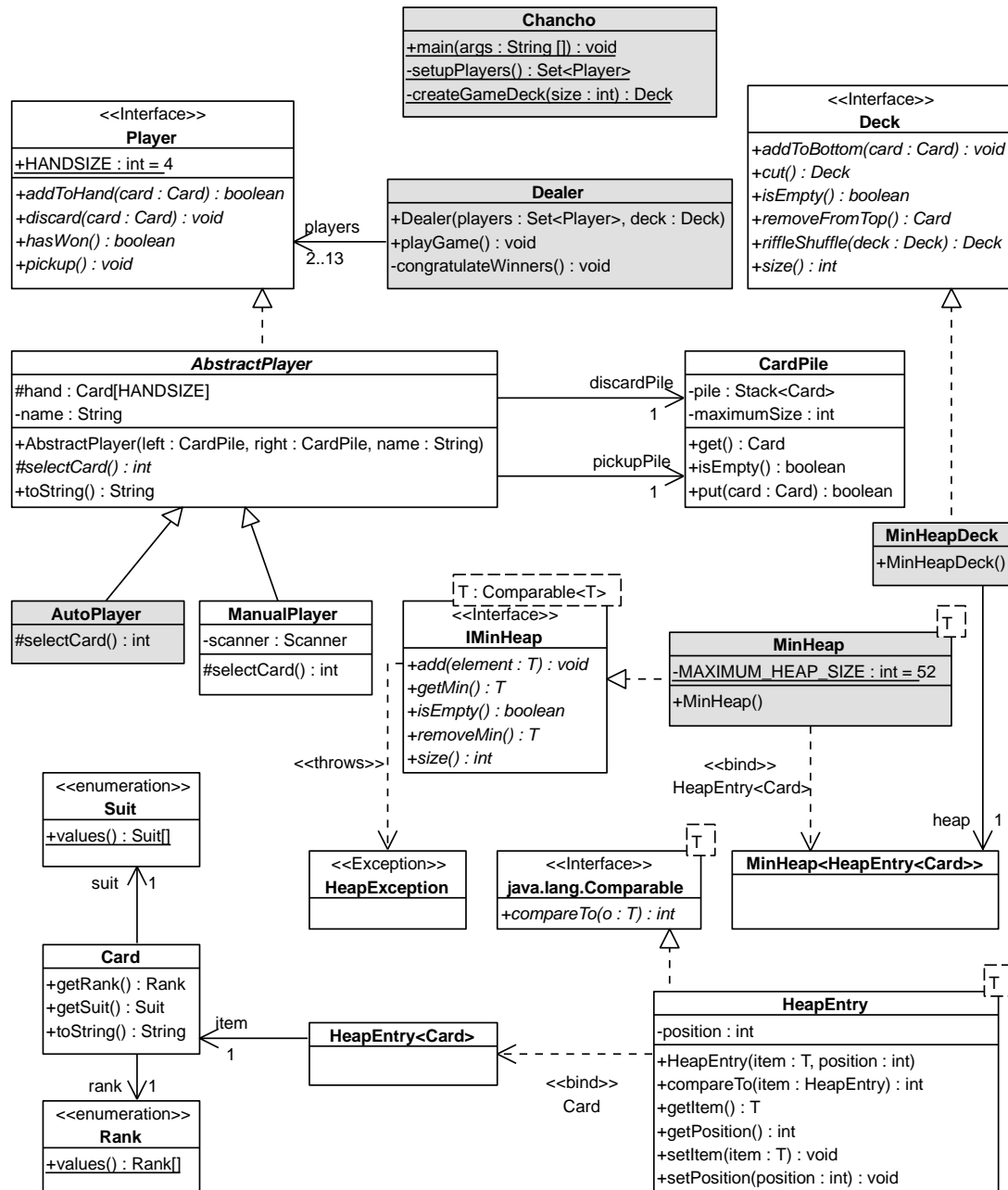


Figure 2: UML Class-Diagram

## Detailed Specification

### Task 1: The Min-Heap Abstract Data Type

A deck of playing cards is represented as a min-heap ADT. Write the generic class `MinHeap<T>` that realises the generic interface `IMinHeap<T extends Comparable<T>>` including any auxiliary private methods. You should refer to the provided file *IMinHeap.java* which includes the full specification for your implemented methods.

### Task 2: Deck Shuffling using the Min-Heap

You are provided with a partial implementation of the class `MinHeapDeck` which realises the `Deck` interface. The class makes use of your generic implementation of a min-heap ADT, `MinHeap<T>`, and a wrapper class, `HeapEntry<T>`. The wrapper class associates each card with a ‘position’ on the heap.

For Example, a deck of four cards built incrementally by adding the cards ACE of SPADES, FIVE of CLUBS, ACE of CLUBS *and then* SEVEN of HEARTS would be stored by a `MinHeap<HeapEntry<Card>>` structure shown in Figure 3. Pay close attention to the numbering or ‘position’ of the heap entries.

1. Implement the method `public Deck cut()` which simply divides the current deck into two equally sized decks, returning the top half of the deck and leaving the current deck equal to the bottom half.
2. Implement the method `public Deck riffleShuffle(Deck deck)`. You may assume the parameter `deck` has equal `size()` to the current deck. The operation should create and return a new deck whose cards are the interleaving of the two decks.

For example, taking the deck shown in Figure 3 as the ‘current’ deck, a riffle shuffle with another deck that has been built incrementally by adding the cards TWO of HEARTS, EIGHT of SPADES, QUEEN of CLUBS *and then* KING of CLUBS would result in the deck illustrated in Figure 4.

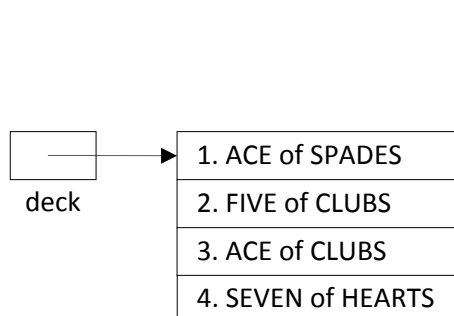


Figure 3: An instance of `MinHeap<HeapEntry<Card>>`

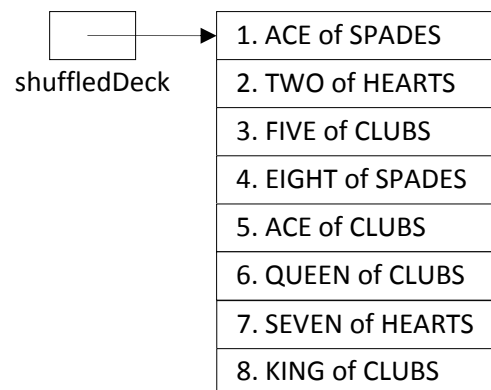


Figure 4: A min-heap structure after a riffle-shuffle

### Task 3: A Computer Card Player

Write the concrete class `AutoPlayer` that extends `AbstractPlayer` and provides an implementation for a card selection strategy.

The method `protected int selectCard()` returns an index into the fixed-size card array which represents the cards held by that player. Your implementation should comply with the following specification.

1. If a single rank appears the most in the player's hand, then a card of *differing* rank is selected randomly<sup>1</sup>.
2. Otherwise, a card is selected randomly from the player's entire hand.

The following table exemplifies this strategy, where we consider a Chanco game with four players.





Array representing the cards held by a player	<code>protected int selectCard()</code>
<div><div><div>A♣</div><div>2♥</div><div>3♣</div><div>2♥</div></div><div>0123</div></div>	<pre>// randomly select a card from // positions 0 and 2 of the array. // i.e. return either 0 or 2</pre>
<div><div><div>J♥</div><div>2♣</div><div>J♣</div><div>2♥</div></div><div>0123</div></div>	<pre>// randomly select a card from the // entire array. // i.e. return either 0, 1, 2 or 3.</pre>
<div><div><div>A♠</div><div>3♣</div><div>2♥</div><div>J♣</div></div><div>0123</div></div>	<pre>// randomly select a card from the // entire array. // i.e. return either 0, 1, 2 or 3.</pre>

---

<sup>1</sup> the class `java.util.Random` provides a method, `public int nextInt(int n)`, which returns a pseudorandom, uniformly distributed `int` value between 0 (inclusive) and `n` (exclusive).

## Task 4: The Card Dealer

Write the class `Dealer`. The constructor of the `Dealer` class is invoked with a set of players and a game `Deck`. You may make the following assumptions about these parameters:

- The `Player` objects have been constructed in such a way that they form a circle, whereby the `discardPile` of one player is the `pickupPile` of another. See Figure 5.
- The number of cards in the game-deck is exactly four times the number of players in the set.
- If a card of a particular rank is included in the game-deck, then cards for all suits of that rank are also included in the game-deck too. e.g., if the card  is included in the game-deck, then the cards ,  and  are also included.

To begin, you will need to write code that deals four cards from the provided game-deck to each player. You must then implement two methods.

`public void playGame()` repeatedly plays a round of the game while no player has declared themselves as a winner. The dealer should check if any player has won at the start of each round, in which case he congratulates the winners and ends the game.

`private void congratulateWinners()` should print to the console a congratulatory message identifying the winners of the game together with their winning hands. In an end-game scenario such as that illustrated in Figure 1, your implementation should produce the following output.

```
The game has been won! Congratulations to:
```

```
Player P1. Cards in hand:
```

```
0) ACE of CLUBS
1) ACE of DIAMONDS
2) ACE of HEARTS
3) ACE of SPADES
```

```
Player P2. Cards in hand:
```

```
0) FIVE of CLUBS
1) FIVE of DIAMONDS
2) FIVE of HEARTS
3) FIVE of SPADES
```

Note: You will need to use the overridden `toString()` method from the `AbstractPlayer` class.

## Task 5: Setting up the Game

Write the method `private static Set<Player> setupPlayers()` in the class `Chanco`. The method should construct four players (three automatic players and one manual player) with four card piles so that they conform to the structure shown in Figure 5. The method should return a set containing the players constructed by the method.

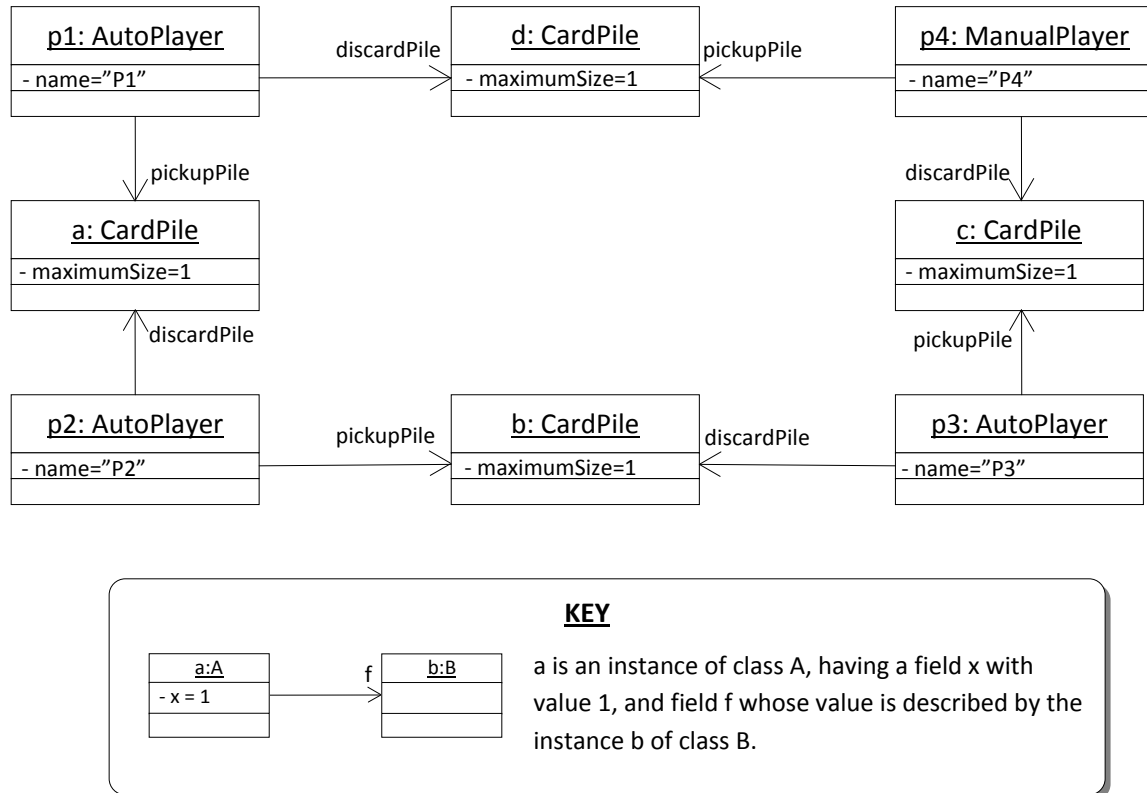


Figure 5: UML Object-Diagram showing the initial setup of the game players