Java: Battle of Cards

Objectives

Your goal is to model a real-world card game in Java using object-oriented techniques, concepts, by using interfaces, abstract classes, using iterators and iterable objects, implementing the *Comparable* and *Comparator* interfaces, overriding *toString*, *hashCode* and *equals*.

Your card game should follow the rules of the *Quartett* card game (referred to sometimes as *"autóskártya"* in Hungarian).

There are many variations of the game, but here's the original instructions of a car themed game.

For 2 or more players. The cards are shuffled and dealt. Each player holds his cards so that only he can see the top card. The player to the left of the dealer reads out any one of the four specifications on the card i. e. Top speed, weight, etc. The other players read out the same specification from their top cards. The player with the highest value takes the top card from the other players (including his own), and puts them at the bottom of his cards. That winner then reads out a specification from his new top card and the game continues. Should any of the specifications be of the same value the top cards are put to one side, and the winner of the next round takes those in addition to the hand. The game ends when any player has no cards left, the winner is the player with most cards.

Requirements

- **Select a *theme* for your card game**. A few popular choices are cars, fighter jets, dinosaurs, football teams, actors, etc. *Use your imagination!*
- **Define your game's rules**. How many players? How many cards dealt per players? Do you want to allow bets?
- **Model your game** using interfaces, classes, etc.
- Use **abstraction, inheritance, composition and encapsulation** where applicable.
- Implement the **Comparable** interface and create and use a **Comparator** implementation at least once.
- Implement the **hashCode** and **equals** methods at least on one class.
- Implement **toString** on classes where it makes sense to pretty print them to the terminal.
- **Print to the standard output *only* in a single place in your application** (see *printStatistics* in *Step 4*).
- Use the **Collections framework** for your implementation.

- **Use Git** to version your application's source code - commit frequently and small, well-defined changes.
- Create a Main class, with a *main* method.

Don't panic! Look at the ***Modelling Walkthrough*** below.

One more thing. ***You cannot use static variables or methods*** (except the *main* method). Create instances of your classes instead.

Modelling Walkthrough

The following steps will walk you through the "modelling" of Blackjack.

This is just a rough example, you need to model your own game on your own. Think before coding!

Step 1.1

Think about the game. How do people play blackjack at a casino? **List the things that come to mind thinking about *walking into a casino to play blackjack***.

- Casino
- **Table**
- **Player**
- **Dealer**
- **Deck** of cards
- **Hand** of cards
- **Card**
- Bet
- *Placing bets*
- *Dealing cards*
- Keeping hand (stand), *drawing more cards* (hit or bust)
- End game
- *Raking in winnings*
- etc.

Choose things to model. There's **no need to model *everything*** - just the bare minimum which seems necessary to build the game.

In the above list nouns are in **bold**, verbs are *underlined and italic*. Nouns will represent classes, interfaces, verbs will represent methods.

Step 2.1

Modelling can be approached in several ways. Two of these are ***top-to-bottom*** or ***bottom-up***. It doesn't matter which one you choose, experiment and see which way works for you better. For this blackjack game we'll start with a bottom-up approach.

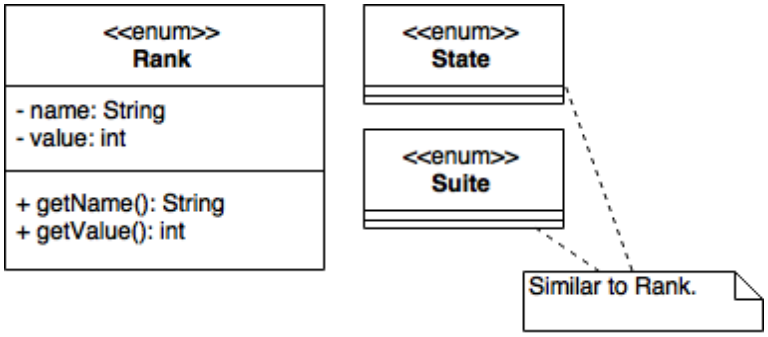
First. What is at **the *core* of blackjack**? You guessed it, ***cards!*** Let's model a card and create a class for it.
First of all. **Pick up a card**, take it into your hards. ***Examine it!***

If you examine more cards you'll **recognize the *pattern*** which are followed by all cards.

Each card has

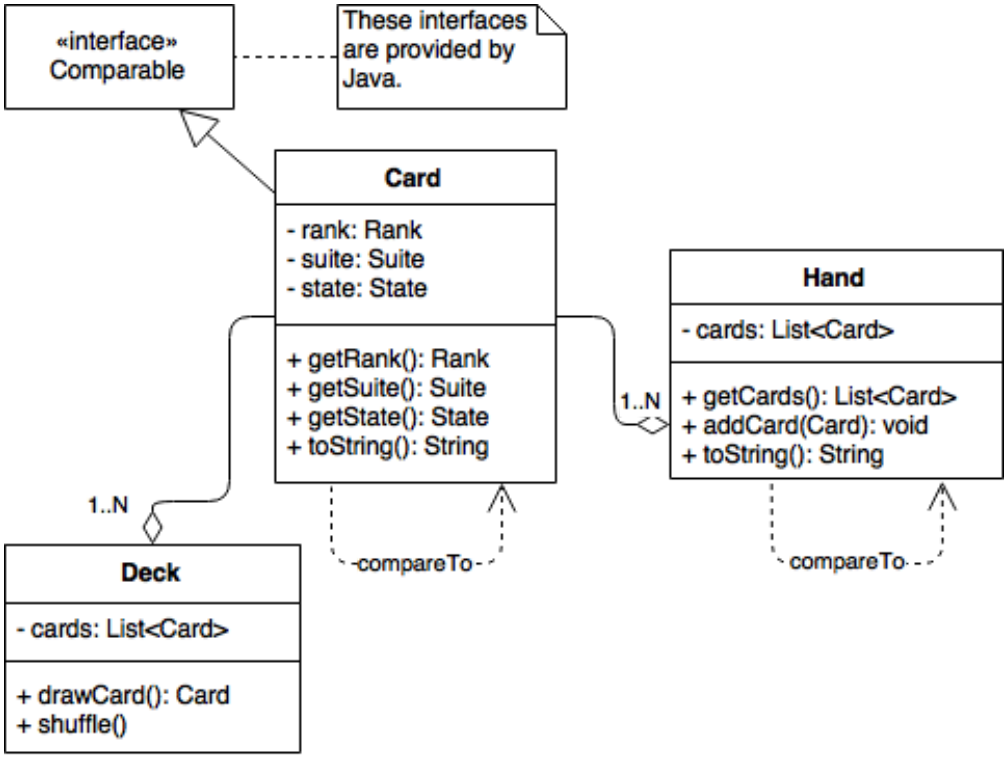
- a *rank* (a value like 10),
- a suite (like Spades),
- and a *state* (face up, face down, etc).

Values like these (finite, countable non-changing things like possible suites of a card) are best modelled with ***Java enums***.



Step 2.2

Based on this we'll create our ***Card* class** … and at the same time we'll define some **more classes** which are based on or using *Card*s. Like a ***Deck*** which consists of a list of 52 cards, and a ***Hand*** (cards held by a player) which consists of 2 or more cards.



Calling *toString* on a Card or a Hand returns the String representation of a Card or a Hand for easier debugging.

Step 2.3

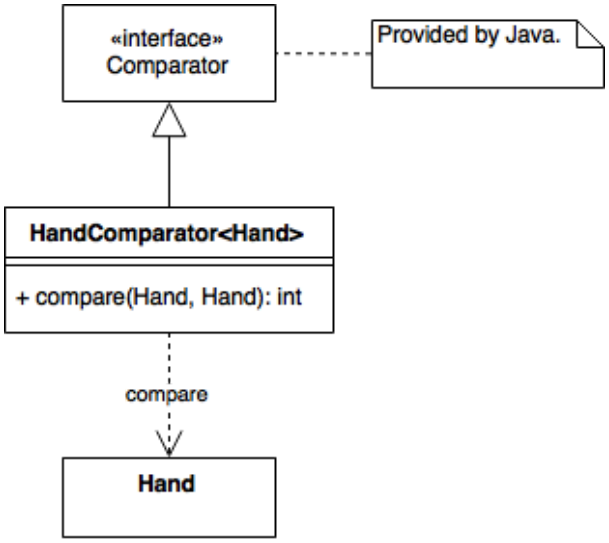
The Card class *implements* the *Comparable* interface provided by Java. Making Card instances comparable to each other.

```
Card card1 = ...
Card card2 = ...
card1.compareTo(card2)
```

This could be used to order cards naturally in a Deck. E.g. in a fresh pack of cards cards could be ordered like, Ace of Spades, Spades 1, Spades 2, …, King of Diamonds.

Step 2.4

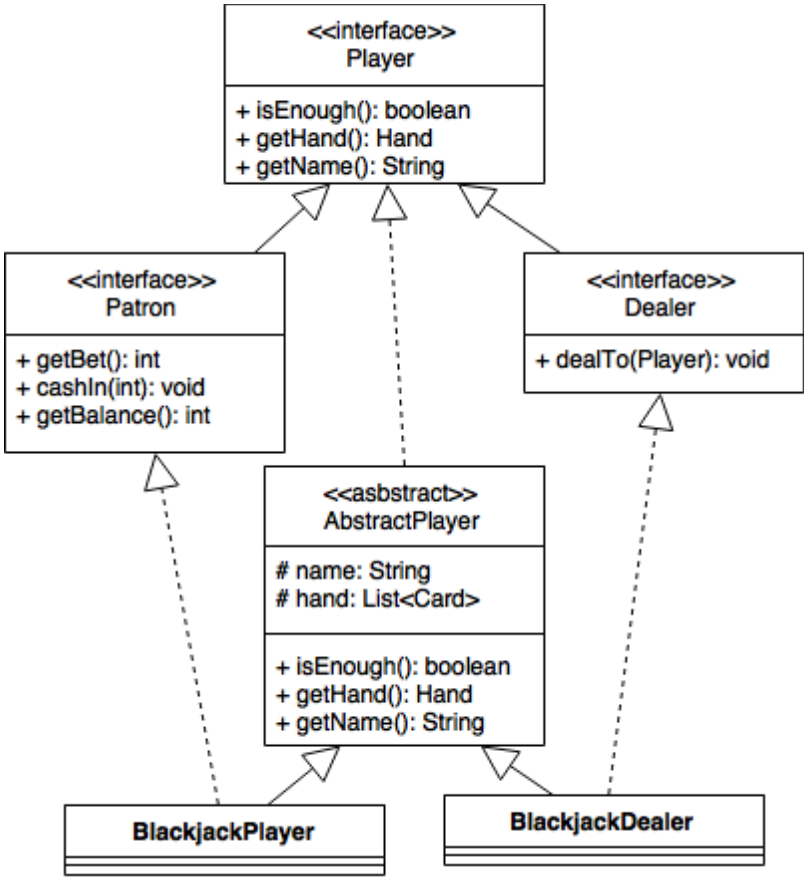
Hand instances are compared using an instances of the *HandComparator* class.



The *HandComparator* implements the *Comparator* interface. It could be used to evaluate players' hands against the dealer's hand of cards to see who wins.

Step 3.1

During a game of blackjack who're going to use these cards? Players of the game obviously. Let's model the concept of a ***Player*** with an **interface**.



Player in the context of blackjack is ambiguous. The dealer is playing too, right? However each player is dealt cards. Let's call a **player playing against the house, the casino a *Patron***. The **player representing the casino will be a *Dealer.***

Patrons and the dealer shares some behaviour. Both of these players have

- a **hand of cards**,
- and can **decide to stand or hit** (draw more cards or stop).

However a ***Dealer*** also can **deal cards** to players (to add a Card to their Hand) - *dealTo(Player)*. Dealers will be able to **deal cards to themselves** (*Dealer* extends the *Player* interface for this purpose).

A ***Patron*** cannot deal cards, but s/he can **place bets and cash in winnings** (and also has a balance).

Concrete realizations of these concepts could be *BlackjackPlayer* and *BlackjackDealer* sharing a common ancestor class *AbstractPlayer*.

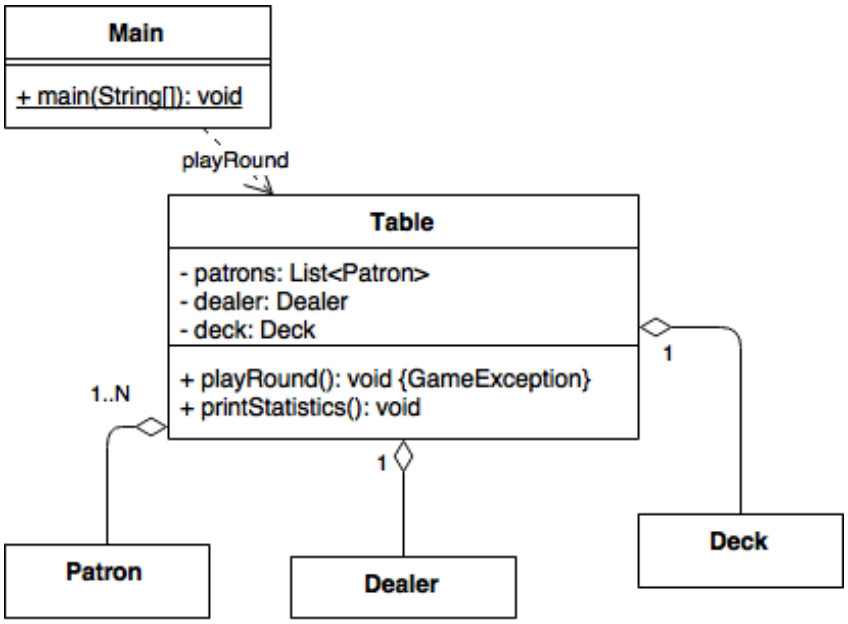
Step 4.1

Going ahead we'll model a *blackjack table*. If you think about it you can come up with the properties of a *Table*.

- **Patron**s can sit there,
- it has one **Dealer**,
- and at least a **Deck** of cards.

And of course ***the game is played at the Table.***

Based on this Let's model a *Table*.



Step 4.2

Calling *playRound* will simulate a game of blackjack. It can throw a *GameException* if something happens while simulating a round of blackjack. E.g. if there are no Patrons sitting at the table it could throw an error saying "no players sitting at the table, game cannot be played".

A pseudo-code for this method would look like this.

```
public void playRound() throws GameException {
        patronsBet();
        dealerDeals();
        patronsTurn();
```

```
        dealerTurn();
        evaluateRound();
}
```

The methods used here could be *private* methods of a hypothetical implementation. Each method simulating some part of the game.

Step 4.3

Pseudo-code for the *patronsBet* would look like this.

```
private void patronsBet() {
    for (Patron patron : patrons) {
        bets.put(patron, patron.getBet());
    }
}
```

Step 4.4

Calling *printStatistics* should print the current status of our blackjack table to standard output: info about the patrons sitting at the table, the current hands, bets placed, etc.

Step 5.1

Now that we've modelled almost every concept/thing that someway or another participates in our blackjack game we have to **define our program's main entry point**, the *Main* class along with the *main* method.

The pseudo-code for this would look like this.

```
public static void main(String[] args) throws GameException {
   Deck deck = ...
   Dealer dealer = ... // dealer is using the Deck instance
   Iterable<Patron> patrons = ...
   Table table = new BlackjackTable(dealer, patrons);
   table.playRound();
   table.printStatistics();
}
```

Summary

Based on the above *Steps* which outline the modelling of Blackjack **create your own game!**

You're completely free to *create your own interfaces, abstract classes and **concrete classes*** to implement the game. The UML diagrams above are giving you only a rough outline and by no means the only way to implement a game, or to organize classes.

Once again, **you're free to experiment, but *keep the Requirements in mind!***