Project 13: Card Game

Due: Sunday, May 3 at 11:59 PM

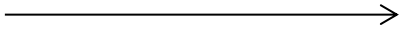
Description: War is a children's card game. It is played with a standard deck of 52 cards (ranks 1 to 13 of four suits called Clubs, Diamonds, Hearts and Spades). Each player is dealt half of the cards. Both players simultaneously place their top card face up. The card with the higher rank wins both cards. Both cards are then moved to the bottom of the winning player's card stack. If there is a tie in rank, a war is declared. Each player puts two additional cards face down and a third card face up. The rank of the third card is used to determine a winner. It is possible to have multiple wars, which is realistically the easiest way to end this game. Many people do not consider War a real game because the outcome is determined by chance alone.

Objectives: Implement the classes described by the UML diagram on the next page. Points will be awarded as follows:

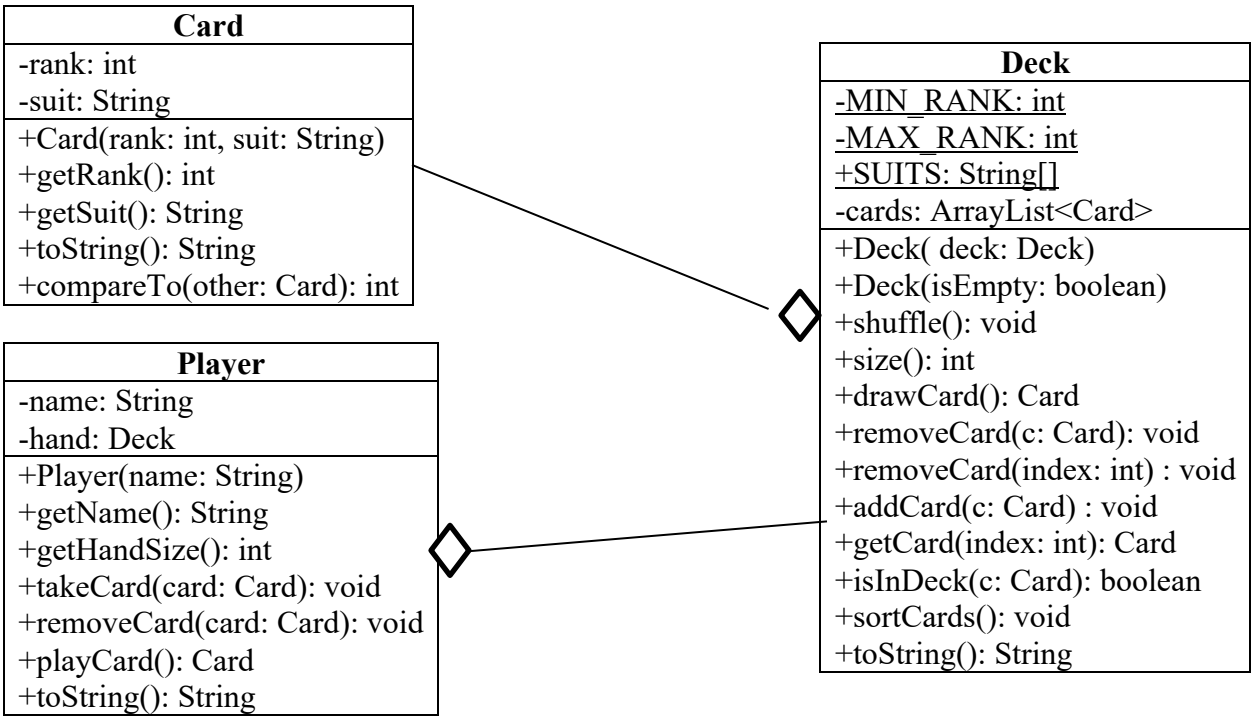
1. (30 points) Implement the Card class.
2. (20 points) Implement the Deck class.
3. (20 points) Implement the Player class.
4. (20 points) Implement two methods in the War class.
5. (10 points) Use meaningful variable names, consistent indentation, and comments to make your code readable.

UML Class Diagrams: Below are UML diagrams for the classes Card, Deck, and Player. These classes represent physical objects and people in a card game. For instance, each Card object has a rank and suit, and each Player object has a name.

The lines with a diamond connecting Card to Deck and Deck to Player indicate aggregation. Aggregation is a relationship between two classes where objects of one class (the class touching the diamond) contain objects of the other class. In this program, Deck object has an ArrayList of Card objects. For Deck objects, the ArrayList represents a physical stack of cards. There is another relationship between the Card and Player classes. The Player class uses the Card class (for parameters, for example), but doesn't contain objects from the class. This relationship is shown in UML using a different symbol that is beyond the scope of this class (the relationship is called Uses and the symbol is a dotted line with an open arrow head, shown below).



To keep the time commitment to this project to a minimum, I'm giving you skeleton code for the Deck and Player classes, and nearly complete code for the War class. This code represents the UML diagrams below. Pay attention to how the UML is translated to code. You will need to perform this translation for the Card class.



War
+main(args: String[]): void -playWar(human: Player, computer: Player, keyboard: Scanner): boolean -handleTie(human: Player, computer: Player, hCard: Card, cCard: Card): void **+moveAllCards(one: Player, deck: deck): void** **+dealHands(one: Player, two: Player): void**

Card Class: Each Card object represents a card from a standard 52-card deck. Note that this class is immutable because there is no way to change the rank or suit of an object after it has been constructed.

Implement toString and compareTo as follows:

- toString: Return the rank and suit as a String. So if the rank is 6 and the suit is Diamonds, "6 of Diamonds" should be returned.
- compareTo: Implementing this method allows an ArrayList of Card objects to be sorted with the sort method of the Collections class. For this to work, you must also add the statement "implements Comparable<Card>" after the class name and before the open brace like this:

  public class Card implements Comparable<Card> {

  compareTo compares the object the method is called on with the given (other) object. The method returns a negative value, 0, or positive value if the implicit object is smaller, equal, or larger than the parameter object.

Write the method so Card objects are sorted first by rank, which is the most convenient ordering for War. An easy way to make this method work is subtracting ranks.

Player Class: Each Player object represents a person playing the game. The class is mutable because Cards can be added and removed from the player's deck. Most of the methods in the Player class call methods from the Deck class.

Implement takeCard, and playCard as follows:

- takeCard: Add the given Card object to the bottom of the Deck.
- playCard: Remmove the Card object from the top of the Deck and return it.

Deck Class: Each Deck object represents a deck of cards. Most of the methods in the Deck class will call methods from the ArrayList class. Many of these methods are very short.

Use the following values for the static constants:

- MIN_RANK: This is the smallest rank. Set it equal to 1.
- MAX_RANK: This is the largest rank. Set it equal to 13. When you are testing the program, you may want to temporarily reduce this to 5, so that you see Wars and repeated Wars.
- SUITS: This String array stores a name for each suit. The usual suit names are Clubs, Diamonds, Hearts, and Spades.
- The Deck class has two constructors, which should initialize objects as follows:
- Deck(isEmpty: boolean): If isEmpty is true, create an empty Deck. If isEmpty is false, create a Deck that contains one Card for each combination of ranks and suits. (Use a pair of nested loops and the static constants.) After constructing the ArrayList and adding the cards, shuffle the ArrayList. (Use a method of the Collections class.)
- Deck(deck: Deck): Create a Deck from a given Deck. Create a copy of the ArrayList before assigning it to the cards field, and shuffle the copy. (Use the keyword "this" to distinguish the field and parameter.)

Finally, implement drawCard as follows:

- drawCard: Check if the ArrayList cards is empty. If not, remove and return the last Card in the list.[1] If so, return null.

War Class: The game logic in the War class, which is provided with these instructions (mostly). You will write just two small methods in the class (the ones in bold in the UML diagram).

If Card, Deck, and Player are written correctly, the War class should run without errors. If the program crashes, you need to do some debugging.

When you see the code, you will see that I've added the special Javadoc documentation. This will create beautiful web pages that describe the code, just like the Java API. To see this documentation, go to Project -> Generate Javadoc. You will see a doc directory appear in your Project folder (it is a sibling of the src directory). Inside this directory, click on index.html and

[1] Removing the last element is faster than removing any other element because none of the remaining elements need to be shifted.

you'll see the documentation. You can (and should) write this type of documentation for all Java programs.

Strategy: The best strategy to use for this project is below.

1. Import the starter code into a src directory of the project.
2. Write the Card class.
3. Write a class called CardDriver to test that the Card class is working properly. Make sure your code runs all of the methods.
4. Write the Deck class.
5. Write a class called DeckDriver to test that the Deck and Card classes are working properly together. Make sure this class runs all of the methods.
6. Write the Player class.
7. Write a class called PlayerDriver to test that Player, Deck and Card are working properly together. Make sure this class runs all of the methods.
8. Run the War program with the Card, Deck, and Player classes. It won't work properly (because two methods are missing). However, it should run. Then write those two methods. When they are complete and correct the game should work. If it doesn't, remember that the errors will be in the code you wrote—the War class is correct already.

The CardDriver, DeckDriver and PlayerDriver classes should not be submitted for grading. Their only purpose was to find problems early in the process.

Upload Code to Zybooks: After you complete each project, you need to upload your source code to Zybooks so it can be graded. If you created the folder structure described earlier, your code is in the folder "Intro to Programming\Projects\Project 14\src". The code is contained in the file Project_14.java. Note that .java files are simply text files that contain Java code. They can be opened with any text editor (e.g., Notepad or TextEdit).

Upload your .java file to Zybooks on the Project 14 assignment page. This requires a few steps. Click the "Submit Assignment" button in the top-right corner. This will reveal a button near the bottom of the page with the text "Choose File." Click this button and browse to the location of your .java file. Finally, click the "Submit for grading" button below the "Choose on hard drive" button.

If Zybooks won't accept the file, make sure you're submitting a .java file, not a .class file. Make sure to see if the output you obtained and the output which I have mentioned match to obtain complete grade.

Looking Beyond 1323/1324: From an object-oriented programming perspective, War is a poorly written class. It consists of only static methods and has a fair amount of duplicate code. For example, many helper methods check whether a given Player object represents the human or computer player and then execute different, but similar, code.

This design has several disadvantages. Chief among them is that the program is difficult to change. If we want to add a feature like a third player, much of the existing code needs to be modified. This is time-consuming and has the potential to introduce bugs.

The design can be dramatically improved by using the tools of inheritance and polymorphism. For example, with inheritance we can create the subclasses HumanPlayer and ComputerPlayer from the Player class. These new classes represent human and computer players.

With polymorphism we can create a method common to HumanPlayer and ComputerPlayer that performs different behavior based on the object type. For instance, a method chooseCard could use a Scanner to ask the user for input when called on a HumanPlayer object. When called on a ComputerPlayer object, the method could use an algorithm to choose a card.

Inheritance and polymorphism are beyond the scope of this class, but they are a core part of the curriculum in CS 2334: Programming Structures and Abstractions. For those of you that continue to study computer science and programming, whether in a class, a job, or in your spare time, you have much to look forward to!