Cards Lab: Day 1 of 5

I will look for responses to the questions or requests in green.

Activity 1: Design and Create a Card Class

**Introduction:**

In this activity, you will complete a Card class that will be used to create card objects.

Think about card games you’ve played. What kinds of information do these games require a card object to

“know”? What kinds of operations do these games require a card object to provide?

**Exploration:**

Now think about implementing a class to represent a playing card. What instance variables should it have?

What methods should it provide?

**Exercises:**

1. Complete the implementation of the provided Card class. You will be required to complete:

a. a constructor that takes two String parameters that represent the card’s rank and suit, and an int

parameter that represents the point value of the card;

b. accessor methods for the card’s rank, suit, and point value;

c. a method to test equality between two card objects (i.e. they are identical cards);

d. Make a CardTester class with a main method that creates 3 Card objects and tests each method for each Card object. Also, include two system.out.println statements, one that uses just a Card object as the parameter, and another right after that uses the toString() method called on the same Card object as the parameter. Notice what they print out.

Paste a snip of your output.

2. Now, back in the Card class, make a toString method to create a String that contains the rank, suit, and point value of the card object. The string should be in the following format:

*rank* of *suit* (point value = *pointValue*)

Use of the @Override annotation before the toString method. The Java @Override annotation can be used to indicate that a method is intended to override a method in a superclass. In this example, the Object class’s toString method is being overridden in the Card class.

If the indicated method doesn’t override a method, then the Java compiler will give an error message.

Here’s a situation where this facility comes in handy. Programmers new to Java often encounter problems

matching headings of overridden methods to the superclass’s original method heading. For example, in the

Weight class below, the tostring method is intended to be invoked when toString is called for a

Weight object.

public class Weight {

private int pounds;

private int ounces;

...

public String tostring(String str) {

return this.pounds + " lb. " + this.ounces + " oz.";

}

...

}

Unfortunately, this doesn't work; the tostring method given above has a different name and a different

signature from the Object class’s toString method. The correct version below has the correct name

toString and no parameter:

public String toString() {

return this.pounds + " lb. " + this.ounces + " oz.";

}

The @Override annotation would cause an error message for the first tostring version to alert the

programmer of the errors.

After you have overridden the toString method in your Card class, re-run your main in CardTester and notice the output from the println statements.

Paste a snip of your output.

What does this tell you about using System.out.println() when you just put the variable name referencing an object as the parameter?

Why was it a good idea for us to override the toString() method in the Card class?

Paste a snip of the toString method:

Activity 2: Initial Design of a Deck Class

**Introduction:**

Think about a deck of cards. How would you describe a deck of cards? When you play card games,

what kinds of operations do these games require a deck to provide?

**Exploration:**

Now consider implementing a class to represent a deck of cards. Describe its instance variables and

methods, general enough to be useful for a variety of card games.

**Exercises:**

1. Write an implementation of the Deck class by coding each of the following:

• Declare instance variables List<Card> cards and int size

• Deck constructor — This constructor receives three arrays as parameters. The arrays contain

the ranks, suits, and point values for each card in the deck. The constructor creates an

ArrayList, and then creates the specified cards and adds them to the list.

For example, if ranks = {"A", "B", "C"}, suits = {"Giraffes", "Lions"},

and values = {2,1,6}, the constructor would create the following cards:

["A", "Giraffes", 2], ["B", "Giraffes", 1], ["C", "Giraffes", 6],

["A", "Lions", 2], ["B", "Lions", 1], ["C", "Lions", 6]

and would add each of them to cards. The parameter size would then be set to the size of

cards, which in this example is 6.

Finally, the constructor should shuffle the deck by calling the shuffle method. Note that you

will not be implementing the shuffle method until Activity 4.

• isEmpty — This method should return true when the size of the deck is 0; false

otherwise.

• size — This method returns the number of cards in the deck that are left to be dealt.

• deal — This method “deals” a card by removing a card from the deck and returning it, if there

are any cards in the deck left to be dealt. It returns null if the deck is empty. There are several

ways of accomplishing this task. Here are two possible algorithms:

**Algorithm 1:** Because the cards are being held in an ArrayList, it would be easy to simply

call the List method that removes an object at a specified index, and return that object.

Removing the object from the end of the list would be more efficient than removing it from the

beginning of the list. Note that the use of this algorithm also requires a separate “discard” list to

keep track of the dealt cards. This is necessary so that the dealt cards can be reshuffled and dealt

again.

**Algorithm 2:** It would be more efficient to leave the cards in the list. Instead of removing the

card, simply decrement the size instance variable and then return the card at size. In this

algorithm, the size instance variable does double duty; it determines which card to “deal” and

it also represents how many cards in the deck are left to be dealt. **This is the algorithm that you**

**should implement.** (The following method may help you to see how this might work.)

Add the following override method to your Deck class, as well. Change variable names as needed:

@Override

public String toString() {

String rtn = "Undealt cards: \n";

for (int k = size - 1; k >= 0; k--) {

rtn = rtn + cards.get(k);

if (k != 0) {

rtn = rtn + ", ";

}

if ((size - k) % 2 == 0) {

// Insert carriage returns so entire deck is visible on console.

rtn = rtn + "\n";

}

}

rtn = rtn + "\nDealt cards: \n";

for (int k = cards.size() - 1; k >= size; k--) {

rtn = rtn + cards.get(k);

if (k != size) {

rtn = rtn + ", ";

}

if ((k - cards.size()) % 2 == 0) {

// Insert carriage returns so entire deck is visible on console.

rtn = rtn + "\n";

}

}

rtn = rtn + "\n";

return rtn;

}

2. Once you have completed the Deck class, create a DeckTester class and add code in the main method to create a Deck object suitable for a game of Blackjack using the guidelines from #3 below. Then test each method for each Deck object 1) before any cares have been dealt, 2) after a card has been dealt, and 3) after all the cards have been dealt.

The output should look something like this:

## The complete deck ##

size: 52

undealt cards:

<...>

dealt cards:

<...>

The deck is empty: false

card(s) dealt:

<...>

size: 51

undealt cards:

<...>

dealt cards:

<...>

The deck is empty: false

card(s) dealt:

<...>

size: 0

undealt cards:

<...>

dealt cards:

<...>

The deck is empty: true

**Questions:**

1. Explain in your own words the relationship between a deck and a card.

2. Consider the deck initialized with the statements below. How many cards does the deck contain?

String[] ranks = {"jack", "queen", "king"};

String[] suits = {"blue", "red"};

int[] pointValues = {11, 12, 13};

Deck d = new Deck(ranks, suits, pointValues);

3. The game of Twenty-One is played with a deck of 52 cards. Ranks run from ace (highest) down to 2

(lowest). Suits are spades, hearts, diamonds, and clubs as in many other games. A face card has point

value 10; an ace has point value 11; point values for 2, …, 10 are 2, …, 10, respectively. Specify the

contents of the ranks, suits, and pointValues arrays so that the statement

Deck d = new Deck(ranks, suits, pointValues);

initializes a deck for a Twenty-One game.

4. Does the order of elements of the ranks, suits, and pointValues arrays matter?

5. Paste a snip of your output.