Purpose

The purpose of this lab is for you to design and write many complex methods, working with arrays and array list objects in many classes.

The problem

A hand of 5 playing cards can be categorized as follows, from least to most valuable:

No pair

The lowest hand.  Five separate cards that do not make any of the hands below.

One pair

Two cards of the same rank, for example two Jacks.

Two pair

Two pairs, for example two 5s and two Kings.

Three of a kind

Three cards of the same rank, for example three 7s.

Straight

Five cards with consecutive ranks, not necessarily of the same suit, such as 7, 8, 9, 10, Jack.  For now, Aces will always be high i.e. Aces rank above Kings, not below 2s.

Flush

Five cards all of the same suit, not necessarily in any order.

Full House

Three of a kind and a pair, for example three 9s and two 4s.

Four of a kind

Four cards of the same rank, for example four Aces.

Straight Flush

Five cards with consecutive ranks, all of the same suit.  Is a straight and a flush

Royal Flush

The best possible hand.  A 10, Jack, Queen, King, Ace, all of the same suit.

You will implement classes that categorize hands of cards.

The Tester class is given to you

You are given a Tester class that tests your classes.  Download the ‘Lab’ program from Canvas, ‘Designing programs’, Example programs.  Tester may not be changed in any way.

Read through main(), see that it creates many hands of cards and prints them out with categories labelled.  Then tests for an invalid hand.

You have already written basic Card and Hand classes, which you will now change and add to for this lab.

You will also write a new class, CountRank.

You will put your classes into the ‘Lab’ program downloaded previously.  You are required to implement the details of each class given below.

Constants

First, a reminder that appropriate constants are required in all classes as appropriate, for clarity and to avoid magic numbers.

The Card class

Represents one single playing card.  For now, Aces will always be high i.e. Aces rank above Kings, not below 2s.

All requirements remain the same, so you can re-use your Card class without changes to begin this lab.

The CountRank class

The idea of counting the ranks in a hand is used by the six categories concerned with rank only, so it will be implemented as a class named CountRank.  The ‘rank categories’ are:

-one pair

-two pair

-three of a kind

-straight

-full house

-four of a kind

The CountRank class counts the ranks of cards in a hand.  Use an integer array to do this, since the playing card ranks 2…Ace are fixed and never change.

For example, for the hand [7♡,  A♠,  2♡,  A♣,  5♢] which is one pair, the count of the different ranks in the hand will be:

See Figure 1 in Lab handout

(Note that ranks of 0 and 1 are not valid, so we simply don’t use these elements in the array.  This clever design idea keeps index the same as rank, for simplicity.)

Required – Exactly and only 1 instance variable:

private int rankCount[];

The following class methods are required:

public CountRank(Hand h)

The constructor sets the counts in the rank array from a hand.

A separate method is required in CountRank for each of the six rank categories, e.g.

public boolean fourOfAKind()

Returns true if any rank count is 4.

and so on.

Just one slight refinement here, that the straight() method must take two parameters:

public boolean straight(int min, int max)

Returns true if consecutive rank counts of 1, at indexes min thru max, inclusive.

Hint:  Hand::straight() must set min and max appropriately when calling this method e.g. for the hand [ K♠, 10♢,  A♢,  Q♣,  J♡], which is a straight:

-Hand::straight() must find the minimum and maximum ranks in the hand.  Here min would be 10 and max 14

-if necessary, Hand::straight() then calls CountRank::straight() with these params.  The rank count array here would be:

See Figure 2 in Lab handout

-CountRank::straight() checks the rank count array between indexes 10 and 14 inclusive, returns true for a straight if all the values are 1

About toString()

Returns a string representation of the rank count array, nicely formatted.  For example, printing the rank count array pictured immediately above would give:

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1}

Should use StringBuilder as you build the big string from the array.

Very useful as you debug your CountRank class.

The Hand class

Represents a hand of 5 cards.  Requirements change for this lab, so you will change and add to your existing Hand class.

A new hand in this lab always starts with five cards.  A hand must be validated during creation, to check that it does not contain duplicate cards.

(Will still use an array list for hand.)

WARNING:  you are not allowed to sort the hand!  Do not submit a solution that uses the Java library Collections class

Required – Exactly and only 1 instance variable:

private ArrayList<Card> cards;

The following class methods are required:

public Hand(Card first, Card second, Card third, Card fourth, Card fifth)

The constructor creates a full hand object from the 5 card objects seen in the method header given here.

Must print an appropriate error message and exit the program if the hand contains any duplicate cards.  (Hint: write a method in the hand class that uses a 2-dimensional array to validate a hand.  The constructor calls this method.)

public Card getCard(int i)

Returns the card at index i from a hand.  (Note: does not remove the card from the hand.)

public int category()

Returns the category of a hand.

Important:  is required to call a separate method in Hand for each of the ten hand categories as it does this i.e.

public int category()  
{  
    int category = NO\_PAIR;

    if (royalFlush())  
        category = ROYAL\_FLUSH;  
    else if (straightFlush())  
        category = STRAIGHT\_FLUSH;  
    else if (fourOfAKind())  
        category = FOUR\_OF\_A\_KIND;  
    . . .

    return category;  
}

So a separate method is required in Hand for each of the ten hand categories, e.g.

public boolean royalFlush()

Returns true if this hand is a royal flush.

Then the six rank category methods in Hand are required to call the equivalent method in the RankCount class e.g.

public boolean onePair()

Returns true if this hand has exactly one pair.

Required to call the CountRank::onePair() method as it does this.  Here’s the Hand method:

public boolean onePair()  
{  
    CountRank cr = new CountRank(this);

    // check cr for exactly one pair  
    return cr.onePair();  
}

Some additional private helper methods will be useful.

private boolean hasAce()

Returns true if hand contains at least one Ace.

private boolean findRank(int rank)

Returns true if the rank is in the hand.

You may add any other methods that you find useful.

About toString()

No special formatting required, so simply:

public String toString()  
{  
    return cards.toString();  
}

**Extra credit**

When the required work above is finished, you can earn 20% extra credit by having your program work with Aces low or high.

Surprisingly, only two methods need to be changed!  Some major hints:

-first, the key insight is to continue to represent all Aces internally as value 14/ACE\_HIGH

-in the card class constructor, now allow the rank parameter coming in to the method to have the new value 1/ACE\_LOW.  Do this by simply changing this value to 14/ACE\_HIGH before using it to set the rank instance variable

-in Hand::straight(), first set min and max from the hand as usual.  Then changes are required, to handle the new special case of having a 2 and an Ace.  (Here, min would be 2 and max 14.)  Given Ace, 2, we now have to check for the rest of an Aces low straight, Ace, 2, 3, 4, 5.  There’s a clever way to do this by updating min and max here.  Then can call CountRank::straight() as usual