CMP-5015Y Assignment 2 (Java)

100263247 (uxk18qau)

Tue, 4 Feb 2020 17:16

PDF prepared using PASS version 1.17 running on Windows 10 10.0 (amd64).

☑ I agree that by submitting a PDF generated by PASS I am confirming that I have checked the PDF and that it correctly represents my submission.

Contents

Card.java

```
1  package question1;
2
3  import java.io.*;
4  import java.util.Comparator;
5  import java.util.Random;
6  import java.util.ArrayList;
7
8  public class Card implements Comparable<Card>, Serializable {
9      private Rank rank;//rank attribute
10     private Suit suit;//suit attribute
11     private static transient final long serialVersionUID = 100263247L;
12     public static void main(String[] args) {
13         Card a = new Card(Rank.TEN,Suit.CLUBS);
14         Card b = new Card(Rank.QUEEN,Suit.DIAMONDS);
15         String afn = write(a);//testing of serialization
16         String bfn = write(b);
17         Card c = read(afn);//testing deserialization
18         Card d = read(bfn);
19         System.out.println("Card a: " +a.toString());
20         System.out.println("Card b: " +b.toString());
21         assert c != null;
22         System.out.println("Card c: " +c.toString());
23         assert d != null;
24         System.out.println("Card d: " +d.toString());
25         System.out.println("Differance between a and b: " + difference(a,b));//test
               differance
           System.out.println("Differance between value of a and b: " + differenceValue(
               a,b));//test differance in value
27         System.out.println("selectTest");
28         selectTest(randomCard());//test select test
29     }
30     /**
31      * The rank enum contains all the possible rank values
32      * TWO-ACE
33      */
34     public enum Rank {
35         TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE;
36         private static Rank[] v = values();
37         private static int[] value = {2,3,4,5,6,7,8,9,10,10,10,10,11};
38         /**
39          * @return the previous enum value in the list
40          */
41         public Rank getPrevious(){
42             if (this == TWO){
43                 return ACE;
44             } else {
45                 return v[(this.ordinal()-1)];
46             }
47         }
48         /**
49          * ADDED
50          * helps with getting which ranks can be played
51          * @return the next enum value in the list
52          */
53         public Rank getNext(){
54             if (this == ACE){
55                 return TWO;
56             } else {
57                 return v[(this.ordinal()+1)];
58             }
59         }
```

```
61          /**
             * @return returns the integer value of the card
             */
63          public int getValue(){
                return value[this.ordinal()];
65          }
        }
67      /**
         * Suit enum containing all the suits
69       */
        public enum Suit {
71          CLUBS, DIAMONDS, HEARTS, SPADES;
            private static Suit[] v = values();
73          /**
             * @return a randomly selected suit
75           */
            static Suit getRandom(){
77              Random random = new Random();
                return v[random.nextInt(4)];
79          }
        }
81      /**
         * Constructor
83       * @param rank the rank the card is
         * @param suit the suit the card is
85       */
        public Card(Rank rank,Suit suit){
87          this.rank=rank;
            this.suit=suit;
89      }
        //accessors methods
91      /**
         * @return the rank of the card
93       */
        public Rank getRank(){
95          return rank;
        }
97      /**
         * @return the suit of the card
99       */
        public Suit getSuit(){
101         return suit;
        }
103     //to string
        /**
105      * toString method of the card
         * @return a string with of rank and suit
107      */
        @Override
109     public String toString(){
            return "The " + getRank() + " of " +getSuit();
111     }
        //methods
113     /**
         * Card Compare
115      * Comparison by rank then by suit
         * @param o other card to compare this card to
117      * @return 1 if greater,0 if the same,-1 if less
         */
119     @Override
        public int compareTo(Card o) {
121             int r = (this.getRank().ordinal())-(o.getRank().ordinal());
                if (r==0){
```

```
123                 int s = (this.getSuit().ordinal())-(o.getSuit().ordinal());
                    if (s==0){
125                     return s;
                    } else if (s>0){
127                     return -1;
                    } else {
129                     return 1;
                    }
131             } else if (r>0){
                    return 1;
133             } else {
                    return -1;
135             }
            }
137     /**
         * Find the difference in ranks between Cards A and B
139      * @param A first Card
         * @param B second Card
141      * @return the difference in ranks between two cards
         */
143     public static int difference(Card A, Card B){//returns the difference in ranks
            between two cards
            return Math.abs((A.rank.ordinal())-(B.rank.ordinal()));
145     }
        /**
147      * find the difference in value between the ranks of Cards A and B
         * @param A first Cards
149      * @param B second Card
         * @return the difference between the value of the Ranks of two cards
151      */
        public static int differenceValue(Card A, Card B){// returns the difference in
            values between two cards
153         return Math.abs(A.rank.getValue()-B.rank.getValue());
        }
155     //Comparator classes
        public static class CompareAscending implements Comparator<Card>{
157         /**
             * Rank comparison between c1 and c2
159          * @param c1 first card
             * @param c2 second rank
161          * @return 1 if c1>c2 0 if they are equal -1 if c1<c2
             */
163         @Override
            public int compare(Card c1, Card c2) {
165             int n = (c1.getRank().ordinal())-(c2.getRank().ordinal());
                int r = 0;
167             if(n>0){
                    r = -1;
169             } else if(n<0){
                    r = 1;
171             }
                return r;
173         }
        }
175     public static class CompareSuit implements Comparator<Card>{
            /**
177          * Comparison between c1 and c2
             * @param c1 first card
179          * @param c2 second card
             * @return 1 if c1>c2 0 if they are equal -1 if c1<c2
181          */
            @Override
183         public int compare(Card c1, Card c2) {
```

```
            int n = (c1.getSuit().ordinal())-(c2.getSuit().ordinal());
185         int r = 0;
            if(n>0){
187             r = -1;
            } else if(n<0){
189             r = 1;
            }
191         return r;
        }
193 }
    /**
195  * creates 3 random Cards and compares them with lamdas to the card passed in
     * @param a card to test
197  */
    static void selectTest(Card a){
199     System.out.println(a.toString());
        ArrayList<Card> cards = new ArrayList<>();
201     for (int i = 0; i <3;i++){
            cards.add(randomCard());
203     }
        Card.CompareAscending RankObject = new Card.CompareAscending();
205     Card.CompareSuit SuitObject = new Card.CompareSuit();
        cards.forEach(n -> System.out.println("  " + n.toString() +":\n    RANK: " +
           RankObject.compare(a,n) + "\n    SUIT: " +SuitObject.compare(a,n) + "\n
              CARD: " + a.compareTo(n)));
207 }
    //serialsation
209 /**
     *  Writes out a card to a file
211  * @param card card to serialize
     * @return the fileName of where the card is saved
213  */
    public static String write(Card card){
215     String fileName = "/" + card.toString() + ".ser";
        try {
217         ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(new
              File(fileName)));
            out.writeObject(card);
219         out.close();
            System.out.println("Serialized data is saved");
221     } catch (IOException i) {
            i.printStackTrace();
223     }
        return fileName;
225 }
    /**
227  * reads in a card from a file (deserialization)
     * @param fileName File of the card
229  * @return the Card
     */
231 public static Card read(String fileName) {
        try {
233         ObjectInputStream in = new ObjectInputStream(new FileInputStream(fileName
              ));
            return (Card) in.readObject();
235     } catch (IOException i) {
            i.printStackTrace();
237         return null;
        } catch (ClassNotFoundException c) {
239         System.out.println("Card class not found");
            c.printStackTrace();
241         return null;
        }
```

```
243     }
        /**
245      * ADDED
         * For select test to generate random cards
247      * Creates a random Card
         * @return a card of random suit and rank
249      */
        public static Card randomCard(){
251         Random random = new Random();
            Rank[] v = Card.Rank.values();
253         return new Card(v[random.nextInt(13)],Suit.getRandom());
        }
255 }
```

Deck.java

```
1  package question1;
2
3  import java.io.*;
   import java.util.Iterator;
5  import java.util.NoSuchElementException;
   import java.util.Random;
7  public class Deck implements Iterable<Card>, Serializable {
       private Card[] cards = new Card[52];
9      private int numOfCards = 52;
       private static transient final long serialVersionUID = 100263237L;
11     public static void main(String[] args) {
           Deck deck = new Deck();
13         System.out.println("Odd even iterator=");//demo odd even iterator
           Iterator<Card> iterator = deck.oddEvenIterator();
15         while (iterator.hasNext()){
               Card card = iterator.next();
17             System.out.println("  " + card.toString());
           }
19         System.out.println("\nTo deal Iterator");//demos dealing iterator
           System.out.println(deck.toString());
21         System.out.println("Deal 1 card");
           deck.Deal();
23         System.out.println("Odd even iterator=");//demo odd even iterator
           iterator = deck.oddEvenIterator();
25         while (iterator.hasNext()){
               Card card = iterator.next();
27             System.out.println("  " + card.toString());
           }
29         System.out.println("\nTo deal Iterator");//demos dealing iterator
           System.out.println(deck.toString());
31         deck.shuffle();//demonstrates shuffle
           System.out.println("After shuffle");
33         System.out.println(deck.toString());
           String shuf = "/ShuffledDeck.ser";
35         write(deck,shuf);
           System.out.println("\nNew Deck");
37         deck.newDeck();
           System.out.println(deck.toString());
39         String org = "/orgDeck.ser";
           write(deck,org);
41         System.out.println("Deserialization of shuffle");
           deck = read(shuf);
43         assert deck != null;
           System.out.println(deck.toString());
45         System.out.println("Deserialization of newDeck");
           deck = read(org);
47         System.out.println(deck.toString());
       }
49     /**
        * Constructor
51      * Creates every card in the deck in order
        */
53     public Deck() {
           Card.Rank[] ranks = {Card.Rank.TWO, Card.Rank.THREE, Card.Rank.FOUR, Card.
               Rank.FIVE, Card.Rank.SIX, Card.Rank.SEVEN, Card.Rank.EIGHT, Card.Rank.
               NINE, Card.Rank.TEN, Card.Rank.JACK, Card.Rank.QUEEN, Card.Rank.KING,
               Card.Rank.ACE};
55         Card.Suit[] suits = {Card.Suit.CLUBS, Card.Suit.DIAMONDS, Card.Suit.HEARTS,
               Card.Suit.SPADES};
           for (int s = 0; s < suits.length; s++) {
57             for (int r = 0; r < ranks.length; r++) {
```

```
                cards[s*13+r] = new Card(ranks[r], suits[s]);
            }
        }
    }
    /**
     * @return the number of undelt cards in the deck
     */
    public int size(){//returns number of cards remaining in the deck
        return numOfCards;
    }
    /**
     * reinitialises the deck to the class Deck
     */
    final void newDeck(){
        Card.Rank[] ranks = {Card.Rank.TWO, Card.Rank.THREE, Card.Rank.FOUR, Card.
            Rank.FIVE, Card.Rank.SIX, Card.Rank.SEVEN, Card.Rank.EIGHT, Card.Rank.
            NINE, Card.Rank.TEN, Card.Rank.JACK, Card.Rank.QUEEN, Card.Rank.KING,
            Card.Rank.ACE};
        Card.Suit[] suits = {Card.Suit.CLUBS, Card.Suit.DIAMONDS, Card.Suit.HEARTS,
            Card.Suit.SPADES};
        for (int s = 0; s < suits.length; s++) {
            for (int r = 0; r < ranks.length; r++) {
                cards[s*13+r] = new Card(ranks[r], suits[s]);
            }
        }
    }
    /**
     * @return The dealing Deck Iterator obj
     */
    @Override
    public Iterator<Card> iterator(){
        return new DeckIterator(this);
    }
    /**
     * Traverses the cards in order to be dealt
     * Goes from top to bottom (starts at the card in position 51, goes down)
     */
    class DeckIterator implements Iterator<Card> {
        private int nextCard;//index of next card
        /**
         * init next card to be at the top
         * @param deck
         */
        public DeckIterator(Deck deck) {
            this.nextCard = deck.size() - 1;
        }
        /**
         * @return if there is a next card
         */
        @Override
        public boolean hasNext(){
            return nextCard >= 0;
        }
        /**
         * Decements nextcard
         * @return card at index next card
         */
        @Override
        public Card next(){
            if(!hasNext()) {
                throw new NoSuchElementException();
            }
            return cards[nextCard--];
```

```
117         }
            /**
119          * removes a card from the deck
             */
121         public void remove(){
                numOfCards--;
123             cards[size()] = null;
            }
125     }
        /**
127      * Shuffles the deck
         * goes through every card in the deck and swap with another random card
129      */
        public void shuffle(){
131         Random random = new Random();
            for (int i = 0; i < size(); i++) {
133             int randomIndexToSwap = random.nextInt(size());
                Card temp = cards[randomIndexToSwap];
135             cards[randomIndexToSwap] = cards[i];
                cards[i] = temp;
137         }
        }
139     /**
         * removes the top card from the deck and returns it
141      * @return top card
         */
143     public Card Deal(){
            Card n = null;
145         if (size() > 0){
                numOfCards--;
147             n = cards[size()];
                cards[size()] = null;
149         }
            return n;
151     }
        /**
153      * @return returns the odd even Deck Iterator
         */
155     public Iterator<Card> oddEvenIterator(){
            return new OddEvenIterator(this);
157     }
        /**
159      * traverses the Cards by first going through all the cards in odd positions,
             then the ones in even positions
         */
161     class OddEvenIterator implements Iterator<Card> {
            private int nextCard;
163         private boolean Odds;
            private int decksize;
165         /**
             * Constructor
167          *  Sets next card to the highest index thats odd
             * @param deck deck to iterate
169          */
            public OddEvenIterator(Deck deck) {
171             if (deck.size()%2==0){
                    this.nextCard = deck.size() - 1;
173             } else {
                    this.nextCard = deck.size() - 2;
175             }
                this.decksize = deck.size();
177             Odds = true;
            }
```

```
179         /**
             * @return if the next card is out the index
181          */
            @Override
183         public boolean hasNext(){
                return nextCard >= 0;
185         }
            /**
187          * @return the next card
             */
189         @Override
            public Card next(){
191             if(!hasNext()) {
                    throw new NoSuchElementException();
193             }
                Card card = cards[nextCard];
195             nextCard -= 2;
                if(Odds && nextCard < 0){
197                 if (decksize%2==0){
                        nextCard = decksize - 2;
199                 } else {
                        nextCard = decksize - 1;
201                 }
                    Odds = false;
203             }
                return card;
205         }
        }
207     //serialsation
        /**
209      *  Writes out a deck to a file
         * @param deck deck to serialize
211      * @param file the file to save to
         */
213     public static void write(Deck deck,String file){
            try {
215             ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(new
                    File(file)));
                out.writeObject(deck);
217             out.close();
                System.out.println("Serialized data is saved");
219         } catch (IOException i) {
                i.printStackTrace();
221         }
            //return fileName;
223     }
        /**
225      * reads in a card from a file (deserialization)
         * @param fileName File of the card
227      * @return the Card
         */
229     public static Deck read(String fileName) {
            try {
231             ObjectInputStream in = new ObjectInputStream(new FileInputStream(fileName
                    ));
                return (Deck) in.readObject();
233         } catch (IOException i) {
                i.printStackTrace();
235             return null;
            } catch (ClassNotFoundException c) {
237             System.out.println("Deck class not found");
                c.printStackTrace();
239             return null;
```

```
        }
241     }
        /**
243      * ADDED
         * ToString
245      * @return String of all the cards in the Deck on different lines
         */
247     @Override
        public String toString(){
249         StringBuilder str = new StringBuilder();
            Iterator<Card> iterator = this.iterator();
251         while (iterator.hasNext()){
                Card card = iterator.next();
253             str.append("  ").append(card.toString()).append("\n");
            }
255         return str.toString();
        }
257 }
```

Hand.java

```
1  package question1;
2
3  import java.io.*;
4  import java.util.*;
5  public class Hand implements Iterable<Card>, Serializable {
6      private List<Card> hand; //This is List so I can change the underlying
           implementation easily
7      private transient List<Card> sortedHand;//The sorted list of all the cards in
           hand
8      private transient int[] rankCount = new int[13];
9      private static transient final long serialVersionUID = 100263257L;
10     public static void main(String[] args) {
11         Hand h = new Hand();//testing all the adds
12         h.add(new Card(Card.Rank.TWO,Card.Suit.CLUBS));
13         h.add(new Card(Card.Rank.THREE,Card.Suit.CLUBS));
14         Hand handToAddAdd = new Hand();
15         handToAddAdd.add(new Card(Card.Rank.FOUR,Card.Suit.CLUBS));
16         handToAddAdd.add(new Card(Card.Rank.FIVE,Card.Suit.CLUBS));
17         Hand handToAdd = new Hand(handToAddAdd);
18         handToAdd.add(new Card(Card.Rank.SIX,Card.Suit.CLUBS));
19         handToAdd.add(new Card(Card.Rank.SEVEN,Card.Suit.CLUBS));
20         h.add(handToAdd);
21         ArrayList<Card> collectionToAdd = new ArrayList<>();
22         collectionToAdd.add(new Card(Card.Rank.EIGHT,Card.Suit.CLUBS));
23         collectionToAdd.add(new Card(Card.Rank.NINE,Card.Suit.CLUBS));
24         collectionToAdd.add(new Card(Card.Rank.TEN,Card.Suit.CLUBS));
25         h.add(collectionToAdd);
26         System.out.println(h.toString());
27         String org = "/orginal.ser";//test serialization
28         write(h,org);
29         System.out.println("Is flush: " + h.isFlush());//test is Flush
30         System.out.println("Is straight: " + h.isStraight());//test is straight
31         System.out.println("Hand Value: " + h.handValue());//test hand value
32         System.out.println("\nBefore Removal=");//testing all the different removal
               methods
33         Hand h1 = new Hand();
34         h1.add(new Card(Card.Rank.NINE,Card.Suit.CLUBS));
35         h1.add(new Card(Card.Rank.THREE,Card.Suit.CLUBS));
36         System.out.println("Hand to remove=");
37         System.out.println(h1.toString());
38         h.remove(h1);
39         System.out.println("After Removal=");
40         System.out.println(h.toString());
41         String afterRem = "/afterRem.ser";
42         write(h,afterRem);
43         int index = 1;
44         System.out.println("Index to Remove: " + index);
45         h.remove(index);
46         System.out.println("After Removal=");
47         System.out.println(h.toString());
48         index = 3;
49         System.out.println("Index to Remove: " + index);
50         h.remove(index);
51         System.out.println("After Removal=");
52         System.out.println(h.toString());
53         h.add(new Card(Card.Rank.TEN,Card.Suit.SPADES));
54         System.out.println("Added The TEN of SPADES");
55         System.out.println(h.toString());
56         System.out.println("Is flush: " + h.isFlush());//testing is flush again
57         System.out.println("Is straight: " + h.isStraight());//testing is straight
               again
```

```
        System.out.println("Hand Value: " + h.handValue());//testing hand value again
59      System.out.println("Count Rank TEN: "+ h.countRank(Card.Rank.TEN));//testing
            count rank
        //deserializeation
61      System.out.println("Original");
        h=read(org);
63      assert h != null;
        System.out.println(h.toString());
65      System.out.println("AfterRemoval");
        h=read(afterRem);
67      assert h != null;
        System.out.println(h.toString());
69      System.out.println("\nSorting");//testing sorting
        Hand sort = new Hand();
71      for (int i = 0;i<10;i++){
            sort.add(Card.randomCard());
73      }
        System.out.println(sort.toString());
75      System.out.println("Ascending");
        sort.sortAscending();
77      System.out.println(sort.toStringSorted());
        System.out.println("Decending");
79      sort.sortDescending();
        System.out.println(sort.toStringSorted());
81      System.out.println("Original order=\n" + sort.toString());//shows original
            order of cards after sorting
    }
83  /**
     * Constructor
85   * sets hand to an empty arraylist
     */
87  public Hand(){
        hand = new ArrayList<>();
89  }
    /**
91   * Constructor
     * Creates a new hand with the cards in the arraylist
93   * @param arrayToAdd arraylist that they hand is initialised to
     */
95  public Hand(Card[] arrayToAdd){
        this();
97      add(Arrays.asList(arrayToAdd));
    }
99  /**
     * Constructor
101  * add a hand into the new hand
     * @param handToAdd hand to add into the new hand
103  */
    public Hand(Hand handToAdd){
105     this();
        add(handToAdd);
107 }
    /**
109  * Add a card to a hand
     * @param card card to add to hand
111  */
    public void add(Card card){
113     hand.add(card);
        Card.Rank cardRank = card.getRank();
115     rankCount[cardRank.ordinal()]++;
    }
117 /**
     * Add a collection to the hand
```

```
119         * @param collectionToAdd the collection to add to the hand
            */
121        public void add(Collection<Card> collectionToAdd){
               for (Card card :collectionToAdd){
123                add(card);
               }
125        }
           /**
127         * Add another hand to the current one
            * @param handToAdd hand to add to the current one
129         */
           public void add(Hand handToAdd){
131            add(handToAdd.hand);
           }
133        /**
            * Removes a card from the hand
135         * @param card card to remove
            * @return True if the card was in the hand and False if it wasnt
137         */
           public boolean remove(Card card){
139            if (hand.contains(card)){
                   Card.Rank cardRank = card.getRank();
141                rankCount[cardRank.ordinal()]--;
                   hand.remove(card);
143                return true;
               } else {
145                return false;
               }
147        }
           /**
149         * Removes all the cards int one hand from another hand
            * @param handToRemove hand to remove
151         * @return True if all the cards were in the hand and False if it werent
            */
153        public boolean remove(Hand handToRemove){
               boolean n = true;
155            for (Card card:handToRemove.hand) {
                   boolean rem = remove(card);
157                if (!rem){
                       n=false;
159                }
               }
161            return n;
           }
163        /**
            * removes card in index
165         * @param index the index of the card to remove
            * @throws IndexOutOfBoundsException
167         */
           public void remove(int index) throws IndexOutOfBoundsException{
169            remove(hand.get(index)); //List.get() throws index out of bounds exception if
                    out of range
           }
171        /**
            * @return the hand iterator
173         */
           public Iterator<Card> iterator(){
175            return new HandIterator(this);
           }
177        /**
            * Iterates through the hand in last added first
179         */
           class HandIterator implements Iterator<Card> {
```

```
181         private int nextCard;
            private int size;
183         /**
             *  Initilises next card and size
185          * @param h hand to iterate
             */
187         public HandIterator(Hand h) {
                this.nextCard = 0;
189             this.size = h.hand.size();
            }
191         /**
             * @return if the next card is there
193          */
            @Override
195         public boolean hasNext(){
                return nextCard < size;
197         }
            /**
199          * gets the next card that was added
             * @return the next card
201          */
            @Override
203         public Card next(){
                if(!hasNext()) {
205                 throw new NoSuchElementException();
                }
207             return hand.get(nextCard++);
            }
209     }
        /**
211      *  Writes out a hand to a file
         * @param hand hand to serialize
213      * @param file file to write to
         */
215     public static void write(Hand hand,String file){
            try {
217             ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(new
                    File(file)));
                out.writeObject(hand);
219             out.close();
                System.out.println("Serialized data is saved");
221         } catch (IOException i) {
                i.printStackTrace();
223         }
            //return fileName;
225     }
        /**
227      * reads in a card from a file (deserialization)
         * @param fileName File of the card
229      * @return the Card
         */
231     public static Hand read(String fileName) {
            try {
233             ObjectInputStream in = new ObjectInputStream(new FileInputStream(fileName
                    ));
                return (Hand) in.readObject();
235         } catch (IOException i) {
                i.printStackTrace();
237             return null;
            } catch (ClassNotFoundException c) {
239             System.out.println("Hand class not found");
                c.printStackTrace();
241             return null;
```

```
        }
    }
    /**
     * Sorts the cards in the hand into descending order with Card.compareTo
     */
    public void sortDescending(){
        sortedHand = new ArrayList<>();
        sortedHand.addAll(hand);
        sortedHand.sort(Card::compareTo);
    }
    /**
     * Sort the hand into ascending order by Rank ASC
     */
    public void sortAscending(){
        sortedHand = new ArrayList<>();
        sortedHand.addAll(hand);
        Comparator<Card> CompAsc = new Card.CompareAscending();
        Collections.sort(sortedHand,CompAsc);
    }
    /**
     * Counts the number of cards of rank r in the hand
     * @param r the rank to count
     * @return the number of cards in the hand of rank r
     */
    public int countRank(Card.Rank r){
        int count = 0;
        Iterator<Card> iterator = this.iterator();
        while (iterator.hasNext()){
            Card card = iterator.next();
            if (r.equals(card.getRank())){
                count++;
            }
        }
        return count;
    }
    /**
     * Gets the value of the hand
     * @return the summation of value of all the card ranks in the hand
     */
    public int handValue(){
        int count = 0;
        for (Card.Rank r : Card.Rank.values()) {
            int n = countRank(r);
            if(n > 0){
                int v = r.getValue();
                count = count + (n*v);
            }
        }
        return count;
    }
    /**
     * @return string with each card in the hand on a new line
     */
    @Override
    public String toString(){
        StringBuilder str = new StringBuilder();
        Iterator<Card> iterator = this.iterator();
        while (iterator.hasNext()){
            Card card = iterator.next();
            str.append("  ").append(card.toString()).append("\n");
        }
        return str.toString();
    }
```

```
305     /**
         * Checks if the hand is a flush
307      * @return if the hand is flush True, else False
         */
309     public boolean isFlush(){
            Iterator<Card> iterator = this.iterator();
311         Card.Suit firstSuit = iterator.next().getSuit();
            while (iterator.hasNext()){
313             Card card = iterator.next();
                if (!card.getSuit().equals(firstSuit)){
315                 return false;
                }
317         }
            return true;
319     }
        /**
321      * Gets if the hand is a straight
         * @return true if all the cards in the hand are consecutive ranks
323      */
        public boolean isStraight(){
325         sortDescending();
            Iterator<Card> iterator = this.sortIterator();
327         Card.Rank Prev = null;
            while (iterator.hasNext()){
329             Card card = iterator.next();
                if (Prev != null){
331                 if (!card.getRank().equals(Prev.getPrevious())){
                        return false;
333                 }
                }
335             Prev = card.getRank();
            }
337         return true;
        }
339     /**
         * ADDED
341      * @return the size of the hand
         */
343     public int size(){
            return hand.size();
345     }
        /**
347      * ADDED
         * the hand in order as a string
349      * @return the string with all the cards in the hand in order
         */
351     public String toStringSorted(){
            StringBuilder str = new StringBuilder();
353         if (sortedHand == null){
                sortAscending();
355         }
            Iterator<Card> iterator = this.sortIterator();
357         while (iterator.hasNext()){
                Card card = iterator.next();
359             str.append("  ").append(card.toString()).append("\n");
            }
361         return str.toString();
        }
363     /**
         * ADDED
365      * @return the sorted hand iterator
         */
367     public Iterator<Card> sortIterator(){
```

```
            return new SortIterator();
        }
        /**
         * ADDED
         * Iterates through the sorted hand array
         */
        class SortIterator implements Iterator<Card> {
            private int nextCard;
            /**
             * Constructor
             * Sorts ascending if no sort has been done
             */
            public SortIterator() {
                if(sortedHand.size()!=hand.size()){
                    sortAscending();
                }
                this.nextCard = sortedHand.size() - 1;
            }
            /**
             * @return if there is a next card
             */
            @Override
            public boolean hasNext(){
                return nextCard >= 0;
            }
            /**
             * @return the next card
             */
            @Override
            public Card next(){
                if(!hasNext()) {
                    throw new NoSuchElementException();
                }
                return sortedHand.get(nextCard--);
            }
        }
    }
```

BasicStrategy.java

```
package question2;

import java.util.Iterator;
import java.util.Random;

public class BasicStrategy implements Strategy {
    /**
     * Decides on whether to cheat or not
     * @param b    the bid this player has to follow (i.e the
     * bid prior to this players turn.
     * @param h    The players current hand
     * @return False unless has to cheat
     */
    @Override
    public boolean cheat(Bid b, Hand h) {
        Card.Rank br = b.getRank();
        Iterator<Card> iterator = h.iterator();
        while (iterator.hasNext()){
            Card card = iterator.next();
            if ((card.getRank().getPrevious().equals(br))||(card.getRank().equals(br)
                )||(card.getRank().getNext().equals(br))){
                return false;
            }
        }
        return true;
    }
    /**
     * If Cheating:  play a single card selected randomly
     * If not cheating: always play the maximum number of cards possible of the
         lowest rank possible
     * @param b    the bid the player has to follow.
     * @param h    The players current hand
     * @param cheat true if the Strategy has decided to cheat (by call to cheat())
              *
     * @return The bid to be played
     */
    @Override
    public Bid chooseBid(Bid b, Hand h, boolean cheat) {
        Hand bh = new Hand();
        Card.Rank r;
        if (cheat){
            Random random = new Random();
            int i = random.nextInt(h.size());
            bh.add(h.getIndex(i));
            int rand = random.nextInt(3);
            if (rand== 0){
                r = b.getRank().getPrevious();
            } else if (rand == 1){
                r = b.getRank();
            } else {
                r=b.getRank().getNext();
            }
        }else {
            int highest = h.countRank(b.getRank().getPrevious());
            r = b.getRank().getPrevious();
            if (h.countRank(b.getRank()) > highest){
                highest = h.countRank(b.getRank());
                r = b.getRank();
            }
            if (h.countRank(b.getRank().getNext()) > highest){
                r = b.getRank().getNext();
```

```
            }
            Iterator<Card> iterator = h.iterator();
            while (iterator.hasNext()){
                Card card = iterator.next();
                if (card.getRank() == r){
                    bh.add(card);
                }
            }
        }
        return new Bid(bh,r);
    }
    /**
     * @param h The players current hand
     * @param b the current bid
     * @return True: only when certain they are cheating (based on your own hand)
     */
    @Override
    public boolean callCheat(Hand h, Bid b) {
        int s = b.getCount();
        s+= h.getRankCount(b.getRank().ordinal());
        return s > 4;
    }
}
```

BasicPlayer.java

```
1  package question2;
2
3  public class BasicPlayer implements Player {
4      private Hand h;
5      private Strategy s;
6      private CardGame g;
7      /**
8       * Constructor
9       * @param s strategy
10      * @param g card game
11      */
12     public BasicPlayer(Strategy s, CardGame g) {
13         h=new Hand();
14         setStrategy(s);
15         setGame(g);
16     }
17     /**
18      * Add card to hand
19      * @param c: Card to add
20      */
21     @Override
22     public void addCard(Card c) {
23         h.add(c);
24     }
25     /**
26      * Add hand to current hand
27      * @param h: hand to add
28      */
29     @Override
30     public void addHand(Hand h) {
31         this.h.add(h);
32     }
33     /**
34      * @return number of cards left in hand
35      */
36     @Override
37     public int cardsLeft() {
38         return h.size();
39     }
40     /**
41      * Sets the game
42      * @param g: the player should contain a reference to the game it is playing in
43      */
44     @Override
45     public void setGame(CardGame g) {
46         this.g = g;
47     }
48     /**
49      * Sets the strategy
50      * @param s: the player should contain a reference to its strategy
51      */
52     @Override
53     public void setStrategy(Strategy s) {
54         this.s=s;
55     }
56     /**
57      * chooses the bid for the player and then removes the bid played from the hand
58      * @param b: the last bid accepted by the game. .
59      * @return new bid to be played by the player
60      */
61     @Override
```

```
    public Bid playHand(Bid b) {
63          Bid nb = s.chooseBid(b,h,s.cheat(b,h));
            h.remove(nb.getHand());
65          return nb;
    }
67  /**
     * @param b: the last players bid
69   * @return Whether the player calls cheat or not
     */
71  @Override
    public boolean callCheat(Bid b) {
73          return s.callCheat(h,b);
    }
75  //accessors
    /**
77   * ADDED
     * @return strategy
79   */
    public Strategy getS() {
81          return s;
    }
83  /**
     * ADDED
85   * @return hand
     */
87  public Hand getH() {return h;}
    /**
89   * ADDED
     * @return game
91   */
    public CardGame getG() {return g;}
93  }
```

BasicCheat.java

```
1  package question2;
2
3  import java.util.*;
   public class BasicCheat implements CardGame{
5      private Player[] players;
       private int nosPlayers;
7      public static final int MINPLAYERS=3;
       private int currentPlayer;
9      private Hand discards;
       private Bid currentBid;
11     private boolean notALLComputer = false;
   //     static int correctCallsMade = 0;
13 //     static int incorrectCallsMade = 0;
   //     static int callsAgainstCorrect = 0;
15 //     static int callsAgainstIncorrect = 0;
       // static variable single_instance of type Singleton
17     private static BasicCheat singleInstance = null;
18
19     public static void main(String[] args){
           test(1000,3);
21     }
       /**
23      * my testing function that plays games and prints the percentage that each
            player won
        */
25     public static void test(int numberOfGames,int playerNumber) {
           if (playerNumber<MINPLAYERS){
27             playerNumber=MINPLAYERS;
           }
29         int[] winners = new int[playerNumber];
           BasicCheat cheat;
31         for (int i = 1;i<numberOfGames+1;i++){
               System.out.println("Gamenum: " + i);
33             cheat=new BasicCheat(playerNumber);
               int w = cheat.playGame();
35             winners[w-1]++;
           }
37         for (int i = 0;i<winners.length;i++) {
               if (numberOfGames/100>=1){
39                 double divider = numberOfGames/100;
                   System.out.println((i+1) + ": " +winners[i]/divider + "%");
41             } else {
                   double multplyer = 100/numberOfGames;
43                 System.out.println((i+1) + ": " +winners[i]*multplyer + "%");
               }
45
           }
47         //System.out.println("CorrectCalls: " + correctCallsMade + "\nIncorrectCalls:
                " +incorrectCallsMade + "\nSuccessfulCallsAgainst: " +
               callsAgainstCorrect+"\nFailtedCallsAgainst: " +callsAgainstIncorrect);
       }
49     /**
        * Constructor
51      */
       private BasicCheat(){
53         this(MINPLAYERS);
       }
55     /**
        * Constructor
57      * @param n number of players
        */
```

```
59      private BasicCheat(int n){
            nosPlayers=n;
61          players=new Player[nosPlayers];
            for(int i=0;i<nosPlayers;i++)
63              players[i]=(new BasicPlayer(new BasicStrategy(),this));
            currentBid=new Bid();
65          Card.Rank[] v = Card.Rank.values();
            Random random = new Random();
67          currentBid.setRank(v[random.nextInt(13)]);
            currentPlayer=0;
69          singleInstance = this;
        }
71      /**
         * ADDED
73       * Gets the player
         * @param i index of player
75       * @return Basicplayer of the player at index i
         */
77      public BasicPlayer getPlayer(int i){
            return (BasicPlayer) players[i];
79      }
        /**
81       * ADDED
         * @return current player
83       */
        public int getCurrentPlayer(){
85          return currentPlayer;
        }
87      /**
         * Plays the turn of the current player
89       * @return true
         */
91      @Override
        public boolean playTurn(){
93          //lastBid=currentBid;
            //Ask player for a play,
95          System.out.println("current bid = "+currentBid);
            currentBid=players[currentPlayer].playHand(currentBid);
97          System.out.println("Player bid = "+currentBid);
            //Add hand played to discard pile
99          discards.add(currentBid.getHand());
            //Offer all other players the chance to call cheat
101         boolean cheat=false;
            for(int i=0;i<players.length && !cheat;i++){
103             if(i!=currentPlayer){
                    cheat=players[i].callCheat(currentBid);
105                 if(cheat){
                        System.out.println("Player called cheat by Player "+(i+1));
107                     if (isCheat(currentBid)){
                            //CHEAT CALLED CORRECTLY
109                         players[currentPlayer].addHand(discards);
                            System.out.println("Player " + (currentPlayer+1) + " cheats!"
                                );
111 //                          if (i==1){
    //                              correctCallsMade+=1;
113 //                          }
    //                          if (currentPlayer==1){
115 //                              callsAgainstCorrect +=1;
    //                          }
117                     } else {
                            //CHEAT CALLED INCORRECTLY
119 //                          if (i==1){
    //                              incorrectCallsMade+=1;
```

```
121 //                          }
    //                          if (currentPlayer==1){
123 //                              callsAgainstIncorrect +=1;
    //                          }
125                         System.out.println("Player " + (currentPlayer+ 1) + " Honest"
                                ");
                            currentPlayer=i;
127                         players[currentPlayer].addHand(discards);
                        }
129                     System.out.println("Adding cards to player "+ (currentPlayer+1));
                        //If cheat is called, current bid reset to an empty bid with rank
                             two whatever the outcome
131                     currentBid=new Bid();
                        Card.Rank[] v = Card.Rank.values();
133                     Random random = new Random();
                        currentBid.setRank(v[random.nextInt(13)]);
135                     //Discards now reset to empty
                        discards=new Hand();
137                     for(Player play: players){
                            BasicPlayer p = (BasicPlayer) play;
139                         if (p.getS() instanceof ThinkerStrategy){
                                ThinkerStrategy T = (ThinkerStrategy) p.getS();
141                             T.cheatCalled();
                            } else if (p.getS() instanceof MyStrategy){
143                             MyStrategy M = (MyStrategy) p.getS();
                                M.cheatCalled(currentPlayer);
145                         }
                        }
147                 }
                }
149         }
            if(!cheat){
151         //Go to the next player
                System.out.println("No Cheat Called");
153             currentPlayer=(currentPlayer+1)%nosPlayers;
            }
155         return true;
        }
157     /**
         * Determines if the game has been won by anyone yet
159      * @return -1, if nobody has won or player index if they have
         */
161     public int winner(){
            for(int i=0;i<nosPlayers;i++){
163             if(players[i].cardsLeft()==0)
                    return i;
165         }
            return -1;
167     }
        /**
169      * Initialises the game with PLayer 1 as Thinker and Player 2 as My
         */
171     public void initialise(){
            StrategyFactory sf = new StrategyFactory();
173         players[0].setStrategy(sf.factory("Thinker"));
            players[1].setStrategy(sf.factory("My"));
175         //Create Deck of cards
            Deck d=new Deck();
177         d.shuffle();
            //Deal cards to players
179         Iterator<Card> it=d.iterator();
            int count=0;
181         while(it.hasNext()){
```

```
                players[count%nosPlayers].addCard(it.next());
183             it.remove();
                count++;
185         }
            //Initialise Discards
187         discards=new Hand();
            //Choose first player
189         currentPlayer=0;
            currentBid=new Bid();
191         Card.Rank[] v = Card.Rank.values();
            Random random = new Random();
193         currentBid.setRank(v[random.nextInt(13)]);
            for(Player play: players){
195             BasicPlayer p = (BasicPlayer) play;
                if (p.getS() instanceof HumanStrategy) {
197                 notALLComputer = true;
                    break;
199             }
            }
201


203     }
        /**
205      * Actually plays the game
         * @return the winner
207      */
        public int playGame(){
209         initialise();
            int c=0;
211         int w = -1;
            Scanner in = new Scanner(System.in);
213         boolean finished=false;
            while(!finished){
215             //Play a hand
                System.out.println(" Cheat turn for player "+(currentPlayer+1));
217             playTurn();
//              System.out.println(" Current discards =\n"+discards);
219 //              for (Player p:players){
//                  BasicPlayer bp = (BasicPlayer) p;
221 //                  bp.getH().sortAscending();
//                  System.out.println("New player Hand\n" + bp.getH().toStringSorted()
    );
223 //              }
                c++;
225             System.out.println(" Turn "+c+ " Complete. Press any key to continue or
                    enter Q to quit>");
                if(notALLComputer){
227                 String str=in.nextLine();
                    if(str.equals("Q")||str.equals("q")||str.equals("quit"))
229                     finished=true;
                }
231             w=winner();
                if(w>=0){
233                 System.out.println("The Winner is Player "+(w+1));
                    finished=true;
235             }
            }
237         return w+1;
        }
239     /**
         * Checks if the player is cheating
241      * @param b bid of the player
         * @return true if they are cheating, false if the bid isnt cheating
```

```
243         */
        public static boolean isCheat(Bid b){
245         for(Card c:b.getHand()){
                if(c.getRank()!=b.getRank())
247                 return true;
            }
249         return false;
        }
251     /**
         * ADDED
253      *Gets instance of the BasicCheat
         * @return instance of the basicCheat Game
255      */
        public static BasicCheat getInstance(){
257         if (singleInstance == null){
                singleInstance = new BasicCheat();
259         }
            return singleInstance;
261     }
        /**
263      * ADDED
         * @return numbere of players
265      */
        public int getNosPlayers() {
267         return nosPlayers;
        }
269 }
```

HumanStrategy.java

```
1  package question2;
2
3  import java.util.Iterator;
   import java.util.Scanner;
5
   public class HumanStrategy implements Strategy {
7      /**
        * @param b    the bid this player has to follow (i.e the bid prior to this
            players turn.
9       * @param h    The players current hand
        * @return If the human has to cheat
11      */
       @Override
13     public boolean cheat(Bid b, Hand h) {
           if (hasToCheat(b,h)){
15             System.out.println("You have to cheat");
               return true;
17         } else {
               return false;
19         }
       }
21     /**
        * ADDED
23      * Determines if the human has to cheat
        * @param b    the bid this player has to follow (i.e the bid prior to this
            players turn.
25      * @param h    The players current hand
        * @return if the player has to cheat
27      */
       private boolean hasToCheat(Bid b, Hand h) {
29         Card.Rank br = b.getRank();
           Iterator<Card> iterator = h.iterator();
31         while (iterator.hasNext()){
               Card card = iterator.next();
33             if ((card.getRank().getPrevious().equals(br))||(card.getRank().equals(br)
                   )||(card.getRank().getNext().equals(br))){
                    return false;
35             }
           }
37         return true;
       }
39     /**
        * gets the bid the human wants to play
41      * @param b    the bid the player has to follow.
        * @param h    The players current hand
43      * @param cheat true if the Strategy has decided to cheat (by call to cheat())
        * @return The bid of the human
45      */
       @Override
47     public Bid chooseBid(Bid b, Hand h, boolean cheat) {
           Hand bh = new Hand();
49         Card.Rank r;
           System.out.println("Your hand is");
51         h.sortDescending();
           Iterator<Card> iterator = h.sortIterator();
53         int count = 0;
           while (iterator.hasNext()){
55             Card card = iterator.next();
               System.out.println("  "+ (h.size() - count) +": " + card.toString());
57             count++;
           }
```

```
59          System.out.println("Last bid was: " + b.getCount() + " " +b.getRank());
            //choosing the cards to play
61          System.out.println("Write down the number of the cards you want to play");
            Scanner input = new Scanner(System. in);
63          String inputString = input.nextLine();
            String[] cardNumber = inputString.split(",");
65          for (String cardNum:cardNumber) {
                if (Integer.parseInt(cardNum) > 0 && Integer.parseInt(cardNum) <=h.size()
                    ){
67                  bh.add(h.getSortedIndex(Integer.parseInt(cardNum) - 1));
                }
69          }
            //choosing rank to say
71          int number;
            do {
73              System.out.println("Choose Rank\n0: " +b.getRank().getPrevious()+"\n1: "
                    +b.getRank()+"\n2: " +b.getRank().getNext());
                number = input.nextInt();
75          } while (number > 2 || number <0);
            if (number == 0){
77              r = b.getRank().getPrevious();
            } else if (number == 1){
79              r = b.getRank();
            } else {
81              r = b.getRank().getNext();
            }
83          h.remove(bh);
            return new Bid(bh,r);
85      }
        /**
87       * Asks if the human wants to call cheat or not
         * @param h    The players current hand
89       * @param b the current bid
         * @return if the human wants to call cheat or not
91       */
        @Override
93      public boolean callCheat(Hand h, Bid b) {
            h.sortAscending();
95          System.out.println("Your hand is\n" + h.toStringSorted());
            System.out.println("Last bid was: " + b.getCount() + " " +b.getRank());
97          Scanner input = new Scanner(System. in);
            int number;
99          do {
                System.out.println("Do you want to call cheat?\n0:False\n1:True");
101             number = input.nextInt();
            } while (number != 0 && number != 1);
103         return number != 0;
        }
105 }
```

ThinkerStrategy.java

```
1  package question2;
2
3  import java.util.Iterator;
   import java.util.Random;
5
   public class ThinkerStrategy implements Strategy {
7      private Hand currentKnownDiscards = new Hand();
       private double p;
9      private int[] saidRankCount = new int[13];
       /**
11      * Constructor
        * @param p probability to call cheat
13      */
       public ThinkerStrategy(double p){
15         this.p=p;
       }
17     /**
        * Sets p to 0.8 as this is a relativly sensible probability
19      */
       public ThinkerStrategy(){
21         this(0.1);
       }
23     /**
        * The Thinker should of course cheat if it has to. It should also occasionally
            cheat when it 'doesnt have to.
25      * @param b    the bid this player has to follow (i.e the bid prior to this
            players turn.
        * @param h    The players current hand
27      * @return if its going to cheat
        */
29     @Override
       public boolean cheat(Bid b, Hand h) {
31         if (hasToCheat(b,h)){
               return true;
33         }
           Random random = new Random();
35         int i = random.nextInt(5);
           //I took ocassionally to mean 2/5
37         return i <= 1;
       }
39     /**
        * ADDED
41      * Determines is the player has to cheat
        * @param b previous bid
43      * @param h hand of player
        * @return if the player has to cheat
45      */
       private boolean hasToCheat(Bid b, Hand h) {
47         Card.Rank br = b.getRank();
           Iterator<Card> iterator = h.iterator();
49         while (iterator.hasNext()){
               Card card = iterator.next();
51             if ((card.getRank().getPrevious().equals(br))||(card.getRank().equals(br)
                   )||(card.getRank().getNext().equals(br))){
                   return false;
53             }
           }
55         return true;
       }
57     /**
        *  If cheating, the Thinker should be more likely to choose higher
```

```
59      * cards to discard than low cards. If not cheating, it should usually play all
            its
        * cards but occasionally play a random number.
61      * @param b    the bid the player has to follow.
        * @param h    The players current hand
63      * @param cheat true if the Strategy has decided to cheat (by call to cheat())
        * @return The bid that the computer selects
65      */
    @Override
67  public Bid chooseBid(Bid b, Hand h, boolean cheat) {
        Hand bh = new Hand();
69      Card.Rank r;
        if (cheat){
71          bh.add(randomCard(h));
            Random random = new Random();
73          int rand = random.nextInt(3);
            if (rand== 0){
75              r = b.getRank().getPrevious();
            } else if (rand == 1){
77              r = b.getRank();
            } else {
79              r=b.getRank().getNext();
            }
81      } else {
            int highest = h.countRank(b.getRank().getNext());
83          r = b.getRank().getNext();
            if (h.countRank(b.getRank()) > highest){
85              highest = h.countRank(b.getRank());
                r = b.getRank();
87          }
            if (h.countRank(b.getRank().getPrevious()) > highest){
89              r = b.getRank().getPrevious();
            }
91          //occasionally (2/5) play a random number not all like below
            Random random = new Random();
93          int rand = random.nextInt(5);
            if (rand > 1){
95              Iterator<Card> iterator = h.iterator();
                while (iterator.hasNext()){
97                  Card card = iterator.next();
                    if (card.getRank() == r){
99                      bh.add(card);
                    }
101             }
            } else {
103             rand = random.nextInt(h.countRank(r)) + 1;
                Iterator<Card> iterator = h.iterator();
105             while (iterator.hasNext() && rand > 0){
                    Card card = iterator.next();
107                 if (card.getRank() == r){
                        bh.add(card);
109                     rand--;
                    }
111             }
            }
113     }
        currentKnownDiscards.add(bh);
115     //add bh cards to said rankcount non essential
        return new Bid(bh,r);
117 }
    /**
119  * ADDED
     * more likely to choose higher cards to discard than low cards
```

```
121         * @param h current player hand
            * @return A card that is more likely to be a high rank
123         */
        private Card randomCard(Hand h){
125         h.sortAscending();
            int min = 0;
127         int max = h.size() - 1;
            while (min != max){
129             int mid = (max+min)/2;
                Random random = new Random();
131             int i = random.nextInt(3);
                if (i == 0){//1/3 chance to move the max down
133                 max = mid;
                } else {//2/3 to move the min up
135                 min = mid;
                }
137         }
            return h.getSortedIndex(max);
139     }

141     /**
         * Decides if the thinker wants to call cheat or not
143      * @param h   The players current hand
         * @param b the current bid
145      * @return If the computer is calling cheat
         */
147     @Override
        public boolean callCheat(Hand h, Bid b) {
149         saidRankCount[b.getRank().ordinal()] += b.getCount();
            Hand allKnownPlay = new Hand(h);
151         allKnownPlay.add(currentKnownDiscards);
            //System.out.println("CurrentKnowDiscards=\n" + currentKnownDiscards);
153         int s = b.getCount();
            Card.Rank r = b.getRank();
155         Iterator<Card> iterator = allKnownPlay.iterator();
            while (iterator.hasNext()){
157             Card card = iterator.next();
                if (card.getRank() == r){
159                 s++;
                }
161         }
            if (s>4){
163             System.out.println("ThinkerCallingCheatCosOfKnownPLay");
                return true;
165         } else {
                //use p and saidRankCount
167             Random rand=new Random();
                float rnd = rand.nextFloat();
169             int copyOfSaidRankCount = saidRankCount[b.getRank().ordinal()];
                if (copyOfSaidRankCount > 4){
171                 while(rnd>=p&&copyOfSaidRankCount>4){
                        rnd = rand.nextFloat();
173                     copyOfSaidRankCount--;
                    }
175                 if (copyOfSaidRankCount > 4){
                        System.out.println("ThinkerCallingCheatCosOfSaidRank");
177                     return true;
                    }
179             }
                return false;
181         }
        }
183     /**
```

```
     * ADDED
185  * Called when a player calls cheat to reset the known discard pile.
     */
187 public void cheatCalled(){
        currentKnownDiscards = new Hand();
189     saidRankCount = new int[13];
    }
191 }
```

MyStrategy.java

```
1  package question2;
2
3  import java.util.Iterator;
   import java.util.Random;
5
   public class MyStrategy implements Strategy {
7      private int own = -1;//own player number
       private int playerNum;//number of players
9      private int[] knownDiscardRankCount;//known rank count for cards in discard
       private int[][] prevKnownRankCount;//at the start of that round, what was known
           about each players hand
11     private int[][] currentKnownRankCount;//what is currently known about each
           players hand
       private int[] saidRankCount;//the number that players have said have gone down of
            each rank that round
13     /**
        * Constructor
15      * Sets up all the variables
        */
17     public MyStrategy(){
           BasicCheat x = BasicCheat.getInstance();
19         playerNum = x.getNosPlayers();
           knownDiscardRankCount = new int[13];
21         saidRankCount = new int[13];
           prevKnownRankCount = new int[playerNum][13];
23         currentKnownRankCount = new int[playerNum][13];
       }
25     /**
        * If it has to cheat it will
27      * Has a random chance to cheat that varies depending upon how many cards in hand
        * @param b    the bid this player has to follow (i.e the bid prior to this
            players turn.
29      * @param h    The players current hand
        * @return if the player is going to cheat or not
31      */
       @Override
33     public boolean cheat(Bid b, Hand h) {
           if (hasToCheat(b,h)){//if it has to cheat
35             return true;
           }
37         //my algorithm for whether to cheat or not
           Random rand=new Random();
39         int rnd = rand.nextInt(101);//random number between 1 and 100
           //giving it a chance to cheat
41         return rnd < cheatLessWithLessCardsInHand(h);
       }
43     /**
        * ADDED
45      * Determies is the computer has to cheat
        * @param b previous bid
47      * @param h current hand
        * @return if the computer has to cheat
49      */
       private boolean hasToCheat(Bid b, Hand h) {
51         Card.Rank br = b.getRank();
           Iterator<Card> iterator = h.iterator();//iterate through all the cards in the
               hand
53         while (iterator.hasNext()){
               Card card = iterator.next();
55             if ((card.getRank().getPrevious().equals(br))||(card.getRank().equals(br)
                   )||(card.getRank().getNext().equals(br))){//if the card is able to be
```

```
                played
                return false;
            }
        }
        return true;
    }
    /**
     * ADDED
     * Max is just under 20
     * 10% at dealt quota (52/number of players)
     * Gives percentage likelyhood to cheat
     * @param h hand
     * @return % of how much to cheat
     */
    private int cheatLessWithLessCardsInHand(Hand h){
        double probabilityOfCheat = 0.0;
        if (h.countRank(h.getIndex(0).getRank()) != h.size()){//if cant just play all
            cards and it be true
            if ((h.size()/(52/playerNum))/3 > 1){//if over peek of sin wave
                probabilityOfCheat = 20;//set to 20%
            } else {
                probabilityOfCheat = 20 * Math.sin((h.size()/(52/playerNum)) * Math.
                    PI/6);//set to the sin wave I designed
            }
        }
        return (int) Math.floor(probabilityOfCheat);
    }
    /**
     * If has to cheat play only one card 80% of the time, 2 the rest
     * If cheating but not having too, play rank that it has and pretend to play
        cards it actually has
     * If not cheating will play all its cards it can 90% of the time, rest of the
        time will play one less of the chosen rank
     * @param b    the bid the player has to follow.
     * @param h    The players current hand
     * @param cheat true if the Strategy has decided to cheat (by call to cheat())
     * @return The bid chosen
     */
    @Override
    public Bid chooseBid(Bid b, Hand h, boolean cheat) {
        if (own == -1){//if own player number unknown
            BasicCheat x = BasicCheat.getInstance();
            own=x.getCurrentPlayer();//set own player number
        }
        Hand bh = new Hand();//init new bid hand
        Card.Rank r;//init new rank to say is being played
        Random rand=new Random();
        if (hasToCheat(b,h)) {
            //has to cheat
            //System.out.println("Has to Cheat ---");
            r=getRankToPlayCheating(b);
            //get the highest known number of that rank cards that were in the hand
                of a player at the start of the round
            int highest=0;
            for (int p = 0;p<playerNum;p++){
                if (highest<prevKnownRankCount[p][r.ordinal()]&&p!=own){
                    highest = prevKnownRankCount[p][r.ordinal()];
                }
            }
            int count = 1;//set the number of cards to play to 1
            if (saidRankCount[r.ordinal()]<7&&h.size()>=2&&highest<=2){//if there is
                enough cards in hand, you dont know that anybody will know you are
                cheating and the said rank is low enough
```

```
111                 count++;//add one to the cards to be played
                    if (saidRankCount[r.ordinal()]<2&&h.size()>=3&&highest<=1){//if there
                        is enough cards in hand, you dont know that anybody will know
                        you are cheating and the said rank is low enough
113                     count++;//add another to the cards to be played
                    }
115             }
                bh.add(getFurthestCards(h,count,r));//add the cards to be played to the
                    hand
117         } else {
                if (cheat){
119                 //cheating but not having to
                    //System.out.println("Cheating but not having to===");
121                 r=getRankToPlay(h,b);
                    int quota = h.countRank(r);//get the number of cards to pretend to
                        have
123                 float rnd = rand.nextFloat();
                    if(rnd>=0.9&&quota>1){//90% play full number you can pretend to play
125                     quota--;
                    }
127                 bh.add(getFurthestCards(h,quota,r));//get the cards to be added to
                        hand
                } else {
129                 //not cheating
                    //System.out.println("Not cheating +++");
131                 r=getRankToPlay(h,b);
                    int quota = h.countRank(r);
133                 float rnd = rand.nextFloat();
                    if(rnd>=0.95&&quota>1){//95% of the time add all the cards
135                     quota--;
                    }
137                 Iterator<Card> iterator = h.iterator();
                    while (iterator.hasNext() && quota > 0) {
139                     Card card = iterator.next();
                        if (card.getRank() == r){//iterate through
141                         bh.add(card);//add the cards of the correct rank
                            quota--;
143                     }
                    }
145             }
            }
147         int[] bidRankCount = getRankCountArray(bh);
            for (int i = 0;i<bidRankCount.length;i++){
149             knownDiscardRankCount[i] += bidRankCount[i];//add all the cards to the
                    known discard rank count
            }
151         return new Bid(bh,r);
        }
153     /**
         * ADDED
155      * add up all the ranks around the possible ranks that the player has
         * this gives it more of an edge in calling if people are cheating
157      * Gets the rank with the most adjacent cards so its easier to call cheat
         * @param h hand of hte player
159      * @param b bid of the player
         * @return the rank the player should play
161      */
        private Card.Rank getRankToPlay(Hand h, Bid b){
163         int prevPrevCount = h.countRank(b.getRank().getPrevious().getPrevious());//
                gets the previous previous rank count
            int prevCount = h.countRank(b.getRank().getPrevious());//gets the previous
                rank count
165         int currentCount = h.countRank(b.getRank());//gets the current rank count
```

```
        int nextCount = h.countRank(b.getRank().getNext()); //gets the next rank
            count
167     int nextNextCount = h.countRank(b.getRank().getNext().getNext()); // gets hte
             next next rank count
        int prevSum = 0;
169     int currentSum = 0;
        int nextSum = 0;
171     if (prevCount != 0){
            prevSum = prevPrevCount + prevCount + currentCount;//sums the rank counts
                 of all the ranks around previous
173     }
        if (currentCount != 0){
175         currentSum = prevCount + currentCount + nextCount;//sums all the rank
                counts of all the ranks around current
        }
177     if (nextCount != 0){
            nextSum = currentCount + nextCount + nextNextCount;//sums all the rank
                counts of all the ranks around the next
179     }
        if (nextSum > currentSum && nextSum >= prevSum){
181         return b.getRank().getNext();//returns next if its sum is greater than
                current and greater than or equal to the previous sum
        } else if (currentSum >= prevSum){
183         return b.getRank();//returns current if current is greater than or equal
                to nextsum and greater than or equal to next
        } else {
185         return b.getRank().getPrevious();//if it is the highest return previous
        }
187 }
    /**
189  * ADDED
     * Gets the rank to play when has to cheat using saidRankCount
191  * @param b last bid
     * @return rank to play
193  */
    private Card.Rank getRankToPlayCheating(Bid b){
195     Card.Rank br = b.getRank();
        int saidCountPrev = saidRankCount[br.getPrevious().ordinal()];
197     int saidCountCurr = saidRankCount[br.ordinal()];
        int saidCountNext = saidRankCount[br.getNext().ordinal()];
199     int highestPrev=0;//get the number of known to have been in someones hand
            last cheat call of one down
        for (int p = 0;p<playerNum;p++){
201         if (highestPrev<prevKnownRankCount[p][br.getPrevious().ordinal()]&&p!=own
                ){
                highestPrev = prevKnownRankCount[p][br.getPrevious().ordinal()];
203         }
        }
205     int highestNext=0;//get the number of known to have been in someones hand
            last cheat call of one up
        for (int p = 0;p<playerNum;p++){
207         if (highestNext<prevKnownRankCount[p][br.getNext().ordinal()]&&p!=own){
                highestNext = prevKnownRankCount[p][br.getNext().ordinal()];
209         }
        }
211     int highestCurrent=0;//get the number of known to have been in someones hand
            last cheat call of current rank
        for (int p = 0;p<playerNum;p++){
213         if (highestCurrent<prevKnownRankCount[p][br.ordinal()]&&p!=own){
                highestCurrent = prevKnownRankCount[p][br.ordinal()];
215         }
        }
217     if (highestPrev<4&&highestNext<4&&highestCurrent<4){
```

```
            //can play all
219         if (saidCountPrev<=saidCountCurr&&saidCountPrev<saidCountNext){
                return br.getPrevious();
221         }else if(saidCountNext<=saidCountCurr&&saidCountNext<saidCountPrev){
                return br.getNext();
223         }else{
                return br;
225         }
        } else if (highestPrev==4&&highestNext<4&&highestCurrent<4){
227         //cant play prev without someone possibly knowing
            if(saidCountNext<=saidCountCurr){
229             return br.getNext();
            }else{
231             return br;
            }
233     } else if (highestPrev<4&&highestNext==4&&highestCurrent<4){
            //cant play next without somebody possibly knwing
235         if(saidCountPrev<=saidCountCurr){
                return br.getPrevious();
237         }else{
                return br;
239         }
        } else if (highestPrev<4&&highestNext<4&&highestCurrent==4){
241         //cant play current without somebody possibly knowing
            if(saidCountNext<saidCountPrev){
243             return br.getNext();
            }else{
245             return br.getPrevious();
            }
247     } else if (highestPrev==4&&highestNext==4&&highestCurrent<4){
            return br;//has to pick else somebpody will possibly know
249     } else if (highestPrev<4&&highestNext==4&&highestCurrent==4){
            return br.getPrevious();//has to pick will possibly
251     } else if (highestPrev==4&&highestNext<4&&highestCurrent==4){
            return br.getNext();
253     } else {
            //cant play any so just pick the lowest said rank trying to get out away
                from having to cheat
255         if(saidCountNext<saidCountPrev){
                return br.getNext();
257         }else{
                return br.getPrevious();
259         }
        }
261 }
    /**
263  * ADDED
     * gets the cards furthest away from being played, and tries to cheat them in to
         improve its chances of having to cheat
265  * @param h the hand
     * @param n the number of cards to return
267  * @return a hand of cards to play
     */
269 private Hand getFurthestCards(Hand h, int n, Card.Rank rankToPlay){
        Hand rh= new Hand();
271     Hand dummyHand = new Hand(h);
        Card cardtoPlay = new Card(rankToPlay, Card.Suit.CLUBS);//temp card
273     for (int quotaFull=0;quotaFull<n;quotaFull++){
            Iterator<Card> iterator = dummyHand.iterator();
275         Card.Rank furthestRank = null;
            int distance = 0;
277         while (iterator.hasNext()) {//gets the furthest distance rank from play
                Card card = iterator.next();
```

```
279                 if(Card.difference(card,cardtoPlay)>distance){
                        distance= Card.difference(card,cardtoPlay);
281                     furthestRank = card.getRank();
                    }
283             }
                boolean added = false;
285             iterator = dummyHand.iterator();
                while (iterator.hasNext()&&!added) {
287                 Card card = iterator.next();
                    if(card.getRank() == furthestRank){
289                     rh.add(card);//adds the furthest card to the hand to play
                        dummyHand.remove(card);
291                     added=true;
                    }
293             }
            }
295         return rh;
        }
297     /**
         * ADDED
299      * @param h hand to count
         * @return the rankcount of the hand
301      */
        private int[] getRankCountArray(Hand h){
303         int[] rc = new int[13];
            for(int i = 0;i<13;i++){
305             rc[i] = h.getRankCount(i);
            }
307         return rc;
        }
309     /**
         * Calls cheat if it is the players final card
311      * Calls cheat if the there is more than 4 of of that rank assuming they are
             telling the truth
         * Use said rank to call cheat, but weights up the risk and is more sure if there
              is more cards in the discard pile
313      * @param h   The players current hand
         * @param b the current bid
315      * @return true if the player wants to call cheat
         */
317     @Override
        public boolean callCheat(Hand h, Bid b) {
319         saidRankCount[b.getRank().ordinal()] += b.getCount();//puts the bid into said
                rank count array
            BasicCheat x = BasicCheat.getInstance();
321         subtractionOfRankCount(x.getCurrentPlayer(),b.getCount());
            BasicPlayer p = x.getPlayer(x.getCurrentPlayer());
323         if (p.cardsLeft()==0){
                return true;//calls on players final turn meaning plays have to tell the
                    truth on thier last turn
325         }
            //use the knowndiscards,player hand,currentknowhplayer hands to call cheat
327         int total = b.getCount();
            int rOrd = b.getRank().ordinal();
329         total += knownDiscardRankCount[rOrd];
            total += h.countRank(b.getRank());
331         for (int i = 0;i<playerNum;i++){
                if (i!=x.getCurrentPlayer() && i!=own){
333                 total += currentKnownRankCount[i][rOrd];
                }
335         }
            if (total > 4){
337             return true;
```

```
        }
339         Random rand=new Random();
            float rnd = rand.nextFloat();
341         int copyOfSaidRankCount = saidRankCount[b.getRank().ordinal()];
            if (copyOfSaidRankCount > 4){
343             while(rnd>=beMoreSureWithMoreCardsInDiscard()&&copyOfSaidRankCount>=4){//
                    be more sure on calling cheat
                    rnd = rand.nextFloat();
345                 copyOfSaidRankCount--;
                }
347             return copyOfSaidRankCount > 4;
            }
349         return false;
        }
351     /**
         * ADDED
353      * Makes the probability higher the more cards there are in the discards
         * @return number between 0 and 1 to determine how sure you must be
355      */
        private double beMoreSureWithMoreCardsInDiscard(){
357         double prob;
            int numInDiscard = 0;
359         for (int i = 0;i<13;i++){
                numInDiscard+=knownDiscardRankCount[i]+saidRankCount[i];
361         }
            prob = Math.cos(numInDiscard * Math.PI/104)/5;//set to the cosine wave I
                designed
363         return prob;
        }
365     /**
         * ADDED
367      * Called when a cheat is called in the game to reset all the known info
         * Called to reset all the known hands and arrays
369      * @param p player whos hand the discards were added too
         */
371     public void cheatCalled(int p){
            if (p!=own){
373             int[] old = prevKnownRankCount[p];
                for (int i=0;i<playerNum;i++){
375                 if (i!=p){
                        prevKnownRankCount[i]=currentKnownRankCount[i];
377                 }
                }
379             for(int i=0;i<13;i++){
                    old[i]+=knownDiscardRankCount[i];
381             }
                prevKnownRankCount[p] = old;
383         }
            knownDiscardRankCount = new int[13];
385         saidRankCount = new int[13];
        }
387     /**
         * ADDED
389      * Called each turn to subtract the known rank counts, so that the known cards
             are 100%accurate
         * @param playerNumber player who played cards
391      * @param cardsPlayed The number of cards played
         */
393     private void subtractionOfRankCount(int playerNumber,int cardsPlayed){
            int[] playersRankCount = currentKnownRankCount[playerNumber];
395         for (int i = 0;i<playersRankCount.length;i++){
                if (playersRankCount[i]<=cardsPlayed){
397                 playersRankCount[i]=0;
```

```
                }else{
399                 playersRankCount[i]-=cardsPlayed;
                }
401         }
            currentKnownRankCount[playerNumber]=playersRankCount;
403     }
}
```

StrategyFactory.java

```
package question2;

public class StrategyFactory {
    public Strategy factory(String strat){
        switch (strat){
            case "Human":
                return (new HumanStrategy());
            case "Thinker":
                return(new ThinkerStrategy());
            case "My":
                return (new MyStrategy());
            default:
                return(new BasicStrategy());
        }
    }
}
```

Card2.java

```
package question2;

import java.io.*;
import java.util.Comparator;
import java.util.Random;
import java.util.ArrayList;
public class Card implements Comparable<Card>, Serializable {
    private Rank rank;//rank attribute
    private Suit suit;//suit attribute
    private static transient final long serialVersionUID = 100263247L;
    public static void main(String[] args) {
        Card a = new Card(Rank.TEN,Suit.CLUBS);
        Card b = new Card(Rank.QUEEN,Suit.DIAMONDS);
        String afn = write(a);//testing of serialization
        String bfn = write(b);
        Card c = read(afn);//testing deserialization
        Card d = read(bfn);
        System.out.println("Card a: " +a.toString());
        System.out.println("Card b: " +b.toString());
        assert c != null;
        System.out.println("Card c: " +c.toString());
        assert d != null;
        System.out.println("Card d: " +d.toString());
        System.out.println("Differance between a and b: " + difference(a,b));//test
            differance
        System.out.println("Difference between value of a and b: " + differenceValue(
            a,b));//test differance in value
        System.out.println("selectTest");
        selectTest(randomCard());//test select test
    }
    /**
     * The rank enum contains all the possible rank values
     * TWO-ACE
     */
    public enum Rank {
        TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE;
        private static Rank[] v = values();
        private static int[] value = {2,3,4,5,6,7,8,9,10,10,10,10,11};
        /**
         * @return the previous enum value in the list
         */
        public Rank getPrevious(){
            if (this == TWO){
                return ACE;
            } else {
                return v[(this.ordinal()-1)];
            }
        }
        /**
         * ADDED
         * helps with getting which ranks can be played
         * @return the next enum value in the list
         */
        public Rank getNext(){
            if (this == ACE){
                return TWO;
            } else {
                return v[(this.ordinal()+1)];
            }
        }
        /**
```

```
60          * @return returns the integer value of the card
            */
62          public int getValue(){
                return value[this.ordinal()];
64          }
        }
66      /**
         * Suit enum containing all the suits
68       */
        public enum Suit {
70          CLUBS, DIAMONDS, HEARTS, SPADES;
            private static Suit[] v = values();
72          /**
             * @return a randomly selected suit
74           */
            static Suit getRandom(){
76              Random random = new Random();
                return v[random.nextInt(4)];
78          }
        }
80      /**
         * Constructor
82       * @param rank the rank the card is
         * @param suit the suit the card is
84       */
        public Card(Rank rank, Suit suit){
86          this.rank=rank;
            this.suit=suit;
88      }
        //accessors methods
90      /**
         * @return the rank of the card
92       */
        public Rank getRank(){
94          return rank;
        }
96      /**
         * @return the suit of the card
98       */
        public Suit getSuit(){
100         return suit;
        }
102     //to string
        /**
104      * toString method of the card
         * @return a string with of rank and suit
106      */
        @Override
108     public String toString(){
            return "The " + getRank() + " of " +getSuit();
110     }
        //methods
112     /**
         * Card Compare
114      * Comparison by rank then by suit
         * @param o other card to compare this card to
116      * @return 1 if greater,0 if the same,-1 if less
         */
118     @Override
        public int compareTo(Card o) {
120             int r = (this.getRank().ordinal())-(o.getRank().ordinal());
                if (r==0){
122                 int s = (this.getSuit().ordinal())-(o.getSuit().ordinal());
```

```
                if (s==0){
124                 return s;
                } else if (s>0){
126                 return -1;
                } else {
128                 return 1;
                }
130         } else if (r>0){
                return 1;
132         } else {
                return -1;
134         }
        }
136     /**
         * Find the difference in ranks between Cards A and B
138      * @param A first Card
         * @param B second Card
140      * @return the difference in ranks between two cards
         */
142     public static int difference(Card A, Card B){//returns the difference in ranks
            between two cards
            return Math.abs((A.rank.ordinal())-(B.rank.ordinal()));
144     }
        /**
146      * find the difference in value between the ranks of Cards A and B
         * @param A first Cards
148      * @param B second Card
         * @return the difference between the value of the Ranks of two cards
150      */
        public static int differenceValue(Card A, Card B){// returns the difference in
            values between two cards
152         return Math.abs(A.rank.getValue()-B.rank.getValue());
        }
154     //Comparator classes
        public static class CompareAscending implements Comparator<Card>{
156         /**
             * Rank comparison between c1 and c2
158          * @param c1 first card
             * @param c2 second rank
160          * @return 1 if c1>c2 0 if they are equal -1 if c1<c2
             */
162         @Override
            public int compare(Card c1, Card c2) {
164             int n = (c1.getRank().ordinal())-(c2.getRank().ordinal());
                int r = 0;
166             if(n>0){
                    r = -1;
168             } else if(n<0){
                    r = 1;
170             }
                return r;
172         }
        }
174     public static class CompareSuit implements Comparator<Card>{
            /**
176          * Comparison between c1 and c2
             * @param c1 first card
178          * @param c2 second card
             * @return 1 if c1>c2 0 if they are equal -1 if c1<c2
180          */
            @Override
182         public int compare(Card c1, Card c2) {
                int n = (c1.getSuit().ordinal())-(c2.getSuit().ordinal());
```

```
184             int r = 0;
                if(n>0){
186                 r = -1;
                } else if(n<0){
188                 r = 1;
                }
190             return r;
            }
192     }
        /**
194      * creates 3 random Cards and compares them with lamdas to the card passed in
         * @param a card to test
196      */
        static void selectTest(Card a){
198         System.out.println(a.toString());
            ArrayList<Card> cards = new ArrayList<>();
200         for (int i = 0; i <3;i++){
                cards.add(randomCard());
202         }
            Card.CompareAscending RankObject = new Card.CompareAscending();
204         Card.CompareSuit SuitObject = new Card.CompareSuit();
            cards.forEach(n -> System.out.println("  " + n.toString() +":\n    RANK: " +
               RankObject.compare(a,n) + "\n    SUIT: " +SuitObject.compare(a,n) + "\n
                  CARD: " + a.compareTo(n)));
206     }
        //serialsation
208     /**
         *  Writes out a card to a file
210      * @param card card to serialize
         * @return the fileName of where the card is saved
212      */
        public static String write(Card card){
214         String fileName = "/" + card.toString() + ".ser";
            try {
216             ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(new
                   File(fileName)));
                out.writeObject(card);
218             out.close();
                System.out.println("Serialized data is saved");
220         } catch (IOException i) {
                i.printStackTrace();
222         }
            return fileName;
224     }
        /**
226      * reads in a card from a file (deserialization)
         * @param fileName File of the card
228      * @return the Card
         */
230     public static Card read(String fileName) {
            try {
232             ObjectInputStream in = new ObjectInputStream(new FileInputStream(fileName
                   ));
                return (Card) in.readObject();
234         } catch (IOException i) {
                i.printStackTrace();
236             return null;
            } catch (ClassNotFoundException c) {
238             System.out.println("Card class not found");
                c.printStackTrace();
240             return null;
            }
242     }
```

```
    /**
244  * ADDED
     * For select test to generate random cards
246  * Creates a random Card
     * @return a card of random suit and rank
248  */
    public static Card randomCard(){
250     Random random = new Random();
        Rank[] v = Card.Rank.values();
252     return new Card(v[random.nextInt(13)],Suit.getRandom());
    }
254 /**
     * ADDED
256  * needed by Collections.remove in hand
     * @param obj comparison obj
258  * @return Card
     */
260 @Override
    public boolean equals(Object obj) {
262     return obj instanceof Card && obj != null && ((Card) obj).getRank() == this.
            getRank() && ((Card) obj).getSuit() == this.getSuit();
    }
264 }
```

Deck2.java

```
package question2;

import java.io.*;
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Random;

public class Deck implements Iterable<Card>, Serializable {
    private Card[] cards = new Card[52];
    private int numOfCards = 52;
    private static transient final long serialVersionUID = 100263237L;
    public static void main(String[] args) {
        Deck deck = new Deck();
        System.out.println("Odd even iterator=");//demo odd even iterator
        Iterator<Card> iterator = deck.oddEvenIterator();
        while (iterator.hasNext()){
            Card card = iterator.next();
            System.out.println("  " + card.toString());
        }
        System.out.println("\nTo deal Iterator");//demos dealing iterator
        System.out.println(deck.toString());
        System.out.println("Deal 1 card");
        deck.Deal();
        System.out.println("Odd even iterator=");//demo odd even iterator
        iterator = deck.oddEvenIterator();
        while (iterator.hasNext()){
            Card card = iterator.next();
            System.out.println("  " + card.toString());
        }
        System.out.println("\nTo deal Iterator");//demos dealing iterator
        System.out.println(deck.toString());
        deck.shuffle();//demonstrates shuffle
        System.out.println("After shuffle");
        System.out.println(deck.toString());
        String shuf = "/ShuffledDeck.ser";
        write(deck,shuf);
        System.out.println("\nNew Deck");
        deck.newDeck();
        System.out.println(deck.toString());
        String org = "/orgDeck.ser";
        write(deck,org);
        System.out.println("Deserializeation of shuffle");
        deck = read(shuf);
        assert deck != null;
        System.out.println(deck.toString());
        System.out.println("Deserializeation of newDeck");
        deck = read(org);
        System.out.println(deck.toString());
    }
    /**
     * Constructor
     * Creates every card in the deck in order
     */
    public Deck() {
        Card.Rank[] ranks = {Card.Rank.TWO, Card.Rank.THREE, Card.Rank.FOUR, Card.
            Rank.FIVE, Card.Rank.SIX, Card.Rank.SEVEN, Card.Rank.EIGHT, Card.Rank.
            NINE, Card.Rank.TEN, Card.Rank.JACK, Card.Rank.QUEEN, Card.Rank.KING,
            Card.Rank.ACE};
        Card.Suit[] suits = {Card.Suit.CLUBS, Card.Suit.DIAMONDS, Card.Suit.HEARTS,
            Card.Suit.SPADES};
        for (int s = 0; s < suits.length; s++) {
```

```
58              for (int r = 0; r < ranks.length; r++) {
                    cards[s*13+r] = new Card(ranks[r], suits[s]);
60              }
            }
62      }
        /**
64       * @return the number of undelt cards in the deck
         */
66      public int size(){//returns number of cards remaining in the deck
            return numOfCards;
68      }
        /**
70       * reinitialises the deck to the class Deck
         */
72      final void newDeck(){
            Card.Rank[] ranks = {Card.Rank.TWO, Card.Rank.THREE, Card.Rank.FOUR, Card.
                Rank.FIVE, Card.Rank.SIX, Card.Rank.SEVEN, Card.Rank.EIGHT, Card.Rank.
                NINE, Card.Rank.TEN, Card.Rank.JACK, Card.Rank.QUEEN, Card.Rank.KING,
                Card.Rank.ACE};
74          Card.Suit[] suits = {Card.Suit.CLUBS, Card.Suit.DIAMONDS, Card.Suit.HEARTS,
                Card.Suit.SPADES};
            for (int s = 0; s < suits.length; s++) {
76              for (int r = 0; r < ranks.length; r++) {
                    cards[s*13+r] = new Card(ranks[r], suits[s]);
78              }
            }
80      }
        /**
82       * @return The dealing Deck Iterator obj
         */
84      @Override
        public Iterator<Card> iterator(){
86          return new DeckIterator(this);
        }
88      /**
         * Traverses the cards in order to be dealt
90       * Goes from top to bottom (starts at the card in position 51, goes down)
         */
92      class DeckIterator implements Iterator<Card> {
            private int nextCard;//index of next card
94          /**
             * init next card to be at the top
96           * @param deck
             */
98          public DeckIterator(Deck deck) {
                this.nextCard = deck.size() - 1;
100         }
            /**
102          * @return if there is a next card
             */
104         @Override
            public boolean hasNext(){
106             return nextCard >= 0;
            }
108         /**
             * Decements nextcard
110          * @return card at index next card
             */
112         @Override
            public Card next(){
114             if(!hasNext()) {
                    throw new NoSuchElementException();
116             }
```

```
            return cards[nextCard--];
        }
        /**
         * removes a card from the deck
         */
        public void remove(){
            numOfCards--;
            cards[size()] = null;
        }
    }
    /**
     * Shuffles the deck
     * goes through every card in the deck and swap with another random card
     */
    public void shuffle(){
        Random random = new Random();
        for (int i = 0; i < size(); i++) {
            int randomIndexToSwap = random.nextInt(size());
            Card temp = cards[randomIndexToSwap];
            cards[randomIndexToSwap] = cards[i];
            cards[i] = temp;
        }
    }
    /**
     * removes the top card from the deck and returns it
     * @return top card
     */
    public Card Deal(){
        Card n = null;
        if (size() > 0){
            numOfCards--;
            n = cards[size()];
            cards[size()] = null;
        }
        return n;
    }
    /**
     * @return returns the odd even Deck Iterator
     */
    public Iterator<Card> oddEvenIterator(){
        return new OddEvenIterator(this);
    }
    /**
     * traverses the Cards by first going through all the cards in odd positions,
        then the ones in even positions
     */
    class OddEvenIterator implements Iterator<Card> {
        private int nextCard;
        private boolean Odds;
        private int decksize;
        /**
         * Constructor
         *  Sets next card to the highest index thats odd
         * @param deck deck to iterate
         */
        public OddEvenIterator(Deck deck) {
            if (deck.size()%2==0){
                this.nextCard = deck.size() - 1;
            } else {
                this.nextCard = deck.size() - 2;
            }
            this.decksize = deck.size();
            Odds = true;
```

```
        }
        /**
         * @return if the next card is out the index
         */
        @Override
        public boolean hasNext(){
            return nextCard >= 0;
        }
        /**
         * @return the next card
         */
        @Override
        public Card next(){
            if(!hasNext()) {
                throw new NoSuchElementException();
            }
            Card card = cards[nextCard];
            nextCard -= 2;
            if(Odds && nextCard < 0){
                if (decksize%2==0){
                    nextCard = decksize - 2;
                } else {
                    nextCard = decksize - 1;
                }
                Odds = false;
            }
            return card;
        }
    }
    //serialsation
    /**
     *  Writes out a deck to a file
     * @param deck deck to serialize
     * @param file the file to save to
     */
    public static void write(Deck deck, String file){
        try {
            ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(new
                File(file)));
            out.writeObject(deck);
            out.close();
            System.out.println("Serialized data is saved");
        } catch (IOException i) {
            i.printStackTrace();
        }
        //return fileName;
    }
    /**
     * reads in a card from a file (deserialization)
     * @param fileName File of the card
     * @return the Card
     */
    public static Deck read(String fileName) {
        try {
            ObjectInputStream in = new ObjectInputStream(new FileInputStream(fileName
                ));
            return (Deck) in.readObject();
        } catch (IOException i) {
            i.printStackTrace();
            return null;
        } catch (ClassNotFoundException c) {
            System.out.println("Deck class not found");
            c.printStackTrace();
```

```
240                 return null;
            }
242     }
        /**
244      * ADDED
         * ToString
246      * @return String of all the cards in the Deck on different lines
         */
248     @Override
        public String toString(){
250         StringBuilder str = new StringBuilder();
            Iterator<Card> iterator = this.iterator();
252         while (iterator.hasNext()){
                Card card = iterator.next();
254             str.append("  ").append(card.toString()).append("\n");
            }
256         return str.toString();
        }
258 }
```

Hand2.java

```
package question2;

import java.io.*;
import java.util.*;

public class Hand implements Iterable<Card>, Serializable {
    private List<Card> hand; //This is List so I can change the underlying
        implementation easily
    private transient List<Card> sortedHand;//The sorted list of all the cards in
        hand
    private transient int[] rankCount = new int[13];
    private static transient final long serialVersionUID = 100263257L;
    public static void main(String[] args) {
        Hand h = new Hand();//testing all the adds
        h.add(new Card(Card.Rank.TWO, Card.Suit.CLUBS));
        h.add(new Card(Card.Rank.THREE, Card.Suit.CLUBS));
        Hand handToAddAdd = new Hand();
        handToAddAdd.add(new Card(Card.Rank.FOUR, Card.Suit.CLUBS));
        handToAddAdd.add(new Card(Card.Rank.FIVE, Card.Suit.CLUBS));
        Hand handToAdd = new Hand(handToAddAdd);
        handToAdd.add(new Card(Card.Rank.SIX, Card.Suit.CLUBS));
        handToAdd.add(new Card(Card.Rank.SEVEN, Card.Suit.CLUBS));
        h.add(handToAdd);
        ArrayList<Card> collectionToAdd = new ArrayList<>();
        collectionToAdd.add(new Card(Card.Rank.EIGHT, Card.Suit.CLUBS));
        collectionToAdd.add(new Card(Card.Rank.NINE, Card.Suit.CLUBS));
        collectionToAdd.add(new Card(Card.Rank.TEN, Card.Suit.CLUBS));
        h.add(collectionToAdd);
        System.out.println(h.toString());
        String org = "/orginal.ser";//test serialization
        write(h,org);
        System.out.println("Is flush: " + h.isFlush());//test is Flush
        System.out.println("Is straight: " + h.isStraight());//test is straight
        System.out.println("Hand Value: " + h.handValue());//test hand value
        System.out.println("\nBefore Removal=");//testing all the different removal
            methods
        Hand h1 = new Hand();
        h1.add(new Card(Card.Rank.NINE, Card.Suit.CLUBS));
        h1.add(new Card(Card.Rank.THREE, Card.Suit.CLUBS));
        System.out.println("Hand to remove=");
        System.out.println(h1.toString());
        h.remove(h1);
        System.out.println("After Removal=");
        System.out.println(h.toString());
        String afterRem = "/afterRem.ser";
        write(h,afterRem);
        int index = 1;
        System.out.println("Index to Remove: " + index);
        h.remove(index);
        System.out.println("After Removal=");
        System.out.println(h.toString());
        index = 3;
        System.out.println("Index to Remove: " + index);
        h.remove(index);
        System.out.println("After Removal=");
        System.out.println(h.toString());
        h.add(new Card(Card.Rank.TEN, Card.Suit.SPADES));
        System.out.println("Added The TEN of SPADES");
        System.out.println(h.toString());
        System.out.println("Is flush: " + h.isFlush());//testing is flush again
        System.out.println("Is straight: " + h.isStraight());//testing is straight
```

```
            again
        System.out.println("Hand Value: " + h.handValue());//testing hand value again
60      System.out.println("Count Rank TEN: "+ h.countRank(Card.Rank.TEN));//testing
            count rank
        //deserializeation
62      System.out.println("Original");
        h=read(org);
64      assert h != null;
        System.out.println(h.toString());
66      System.out.println("AfterRemoval");
        h=read(afterRem);
68      assert h != null;
        System.out.println(h.toString());
70      System.out.println("\nSorting");//testing sorting
        Hand sort = new Hand();
72      for (int i = 0;i<10;i++){
            sort.add(Card.randomCard());
74      }
        System.out.println(sort.toString());
76      System.out.println("Ascending");
        sort.sortAscending();
78      System.out.println(sort.toStringSorted());
        System.out.println("Decending");
80      sort.sortDescending();
        System.out.println(sort.toStringSorted());
82      System.out.println("Original order=\n" + sort.toString());//shows original
            order of cards after sorting
    }
84  /**
     * Constructor
86   * sets hand to an empty arraylist
     */
88  public Hand(){
        hand = new ArrayList<>();
90  }
    /**
92   * Constructor
     * Creates a new hand with the cards in the arraylist
94   * @param arrayToAdd arraylist that they hand is initialised to
     */
96  public Hand(Card[] arrayToAdd){
        this();
98      add(Arrays.asList(arrayToAdd));
    }
100 /**
     * Constructor
102  * add a hand into the new hand
     * @param handToAdd hand to add into the new hand
104  */
    public Hand(Hand handToAdd){
106     this();
        add(handToAdd);
108 }
    /**
110  * Add a card to a hand
     * @param card card to add to hand
112  */
    public void add(Card card){
114     hand.add(card);
        Card.Rank cardRank = card.getRank();
116     rankCount[cardRank.ordinal()]++;
    }
118 /**
```

```
     * Add a collection to the hand
120  * @param collectionToAdd the collection to add to the hand
     */
122 public void add(Collection<Card> collectionToAdd){
        for (Card card :collectionToAdd){
124         add(card);
        }
126 }
    /**
128  * Add another hand to the current one
     * @param handToAdd hand to add to the current one
130  */
    public void add(Hand handToAdd){
132     add(handToAdd.hand);
    }
134 /**
     * Removes a card from the hand
136  * @param card card to remove
     * @return True if the card was in the hand and False if it wasnt
138  */
    public boolean remove(Card card){
140     if (hand.contains(card)){
            Card.Rank cardRank = card.getRank();
142         rankCount[cardRank.ordinal()]--;
            hand.remove(card);
144         return true;
        } else {
146         return false;
        }
148 }
    /**
150  * Removes all the cards int one hand from another hand
     * @param handToRemove hand to remove
152  * @return True if all the cards were in the hand and False if it werent
     */
154 public boolean remove(Hand handToRemove){
        boolean n = true;
156     for (Card card:handToRemove.hand) {
            boolean rem = remove(card);
158         if (!rem){
                n=false;
160         }
        }
162     return n;
    }
164 /**
     * removes card in index
166  * @param index the index of the card to remove
     * @throws IndexOutOfBoundsException
168  */
    public void remove(int index) throws IndexOutOfBoundsException{
170     remove(hand.get(index)); //List.get() throws index out of bounds exception if
             out of range
    }
172 /**
     * @return the hand iterator
174  */
    public Iterator<Card> iterator(){
176     return new HandIterator(this);
    }
178 /**
     * Iterates through the hand in last added first
180  */
```

```
    class HandIterator implements Iterator<Card> {
182         private int nextCard;
            private int size;
184         /**
             *  Initilises next card and size
186          * @param h hand to iterate
             */
188         public HandIterator(Hand h) {
                this.nextCard = 0;
190             this.size = h.hand.size();
            }
192         /**
             * @return if the next card is there
194          */
            @Override
196         public boolean hasNext(){
                return nextCard < size;
198         }
            /**
200          * gets the next card that was added
             * @return the next card
202          */
            @Override
204         public Card next(){
                if(!hasNext()) {
206                 throw new NoSuchElementException();
                }
208             return hand.get(nextCard++);
            }
210     }
        /**
212      *  Writes out a hand to a file
         * @param hand hand to serialize
214      * @param file file to write to
         */
216     public static void write(Hand hand,String file){
            try {
218             ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(new
                    File(file)));
                out.writeObject(hand);
220             out.close();
                System.out.println("Serialized data is saved");
222         } catch (IOException i) {
                i.printStackTrace();
224         }
            //return fileName;
226     }
        /**
228      * reads in a card from a file (deserialization)
         * @param fileName File of the card
230      * @return the Card
         */
232     public static Hand read(String fileName) {
            try {
234             ObjectInputStream in = new ObjectInputStream(new FileInputStream(fileName
                    ));
                return (Hand) in.readObject();
236         } catch (IOException i) {
                i.printStackTrace();
238             return null;
            } catch (ClassNotFoundException c) {
240             System.out.println("Hand class not found");
                c.printStackTrace();
```

```
242             return null;
            }
244     }
        /**
246      * Sorts the cards in the hand into descending order with Card.compareTo
         */
248     public void sortDescending(){
            sortedHand = new ArrayList<>();
250         sortedHand.addAll(hand);
            sortedHand.sort(Card::compareTo);
252     }
        /**
254      * Sort the hand into ascending order by Rank ASC
         */
256     public void sortAscending(){
            sortedHand = new ArrayList<>();
258         sortedHand.addAll(hand);
            Comparator<Card> CompAsc = new Card.CompareAscending();
260         Collections.sort(sortedHand,CompAsc);
        }
262     /**
         * Counts the number of cards of rank r in the hand
264      * @param r the rank to count
         * @return the number of cards in the hand of rank r
266      */
        public int countRank(Card.Rank r){
268         int count = 0;
            Iterator<Card> iterator = this.iterator();
270         while (iterator.hasNext()){
                Card card = iterator.next();
272             if (r.equals(card.getRank())){
                    count++;
274             }
            }
276         return count;
        }
278     /**
         * Gets the value of the hand
280      * @return the summation of value of all the card ranks in the hand
         */
282     public int handValue(){
            int count = 0;
284         for (Card.Rank r : Card.Rank.values()) {
                int n = countRank(r);
286             if(n > 0){
                    int v = r.getValue();
288                 count = count + (n*v);
                }
290         }
            return count;
292     }
        /**
294      * @return string with each card in the hand on a new line
         */
296     @Override
        public String toString(){
298         StringBuilder str = new StringBuilder();
            Iterator<Card> iterator = this.iterator();
300         while (iterator.hasNext()){
                Card card = iterator.next();
302             str.append("  ").append(card.toString()).append("\n");
            }
304         return str.toString();
```

```
    }
306 /**
     * Checks if the hand is a flush
308  * @return if the hand is flush True, else False
     */
310 public boolean isFlush(){
        Iterator<Card> iterator = this.iterator();
312     Card.Suit firstSuit = iterator.next().getSuit();
        while (iterator.hasNext()){
314         Card card = iterator.next();
            if (!card.getSuit().equals(firstSuit)){
316             return false;
            }
318     }
        return true;
320 }
    /**
322  * Gets if the hand is a straight
     * @return true if all the cards in the hand are consecutive ranks
324  */
    public boolean isStraight(){
326     sortDescending();
        Iterator<Card> iterator = this.sortIterator();
328     Card.Rank Prev = null;
        while (iterator.hasNext()){
330         Card card = iterator.next();
            if (Prev != null){
332             if (!card.getRank().equals(Prev.getPrevious())){
                    return false;
334             }
            }
336         Prev = card.getRank();
        }
338     return true;
    }
340 /**
     * ADDED
342  * @return the size of the hand
     */
344 public int size(){
        return hand.size();
346 }
    /**
348  * ADDED
     * the hand in order as a string
350  * @return the string with all the cards in the hand in order
     */
352 public String toStringSorted(){
        StringBuilder str = new StringBuilder();
354     if (sortedHand == null){
            sortAscending();
356     }
        Iterator<Card> iterator = this.sortIterator();
358     while (iterator.hasNext()){
            Card card = iterator.next();
360         str.append("  ").append(card.toString()).append("\n");
        }
362     return str.toString();
    }
364 /**
     * ADDED
366  * gets the card in specific index in the hand
     * @param i index
```

```
368         * @return the card at index
            */
370         public Card getIndex(int i){
                return this.hand.get(i);
372         }
            /**
374         * ADDED
            * if cards havent been sorted yet, defaults to sort Asc
376         * @param i index
            * @return card at index of sorted hand
378         */
            public Card getSortedIndex(int i){
380             if(sortedHand.size() == 0){
                    sortAscending();
382             }
                return this.sortedHand.get(i);
384         }
            /**
386         * ADDED
            * @return the sorted hand iterator
388         */
            public Iterator<Card> sortIterator(){
390             return new SortIterator();
            }
392         /**
            * ADDED
394         * Iterates through the sorted hand array
            */
396         class SortIterator implements Iterator<Card> {
                private int nextCard;
398             /**
                 * Constructor
400              * Sorts ascending if no sort has been done
                 */
402             public SortIterator() {
                    if(sortedHand.size()!=hand.size()){
404                     sortAscending();
                    }
406                 this.nextCard = sortedHand.size() - 1;
                }
408             /**
                 * @return if there is a next card
410              */
                @Override
412             public boolean hasNext(){
                    return nextCard >= 0;
414             }
                /**
416              * @return the next card
                 */
418             @Override
                public Card next(){
420                 if(!hasNext()) {
                        throw new NoSuchElementException();
422                 }
                    return sortedHand.get(nextCard--);
424             }
            }
426         /**
            * @param i index of rank
428         * @return the count of the rank at index i
            */
430         public int getRankCount(int i) {
```

```
            return rankCount[i];
        }
    }
```