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

```
package question1;

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("Differance 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)];
            }
        }
        /**
         * @return returns the integer value of the card
         */
        public int getValue(){
            return value[this.ordinal()];
        }
    }
    /**
     * Suit enum containing all the suits
     */
    public enum Suit {
        CLUBS, DIAMONDS, HEARTS, SPADES;
        private static Suit[] v = values();
        /**
         * @return a randomly selected suit
         */
        static Suit getRandom(){
            Random random = new Random();
            return v[random.nextInt(4)];
        }
    }
    /**
     * Constructor
     * @param rank the rank the card is
     * @param suit the suit the card is
     */
    public Card(Rank rank,Suit suit){
        this.rank=rank;
        this.suit=suit;
    }
    //accessors methods
    /**
     * @return the rank of the card
     */
    public Rank getRank(){
        return rank;
    }
    /**
     * @return the suit of the card
     */
    public Suit getSuit(){
        return suit;
    }
    //to string
    /**
     * toString method of the card
     * @return a string with of rank and suit
     */
    @Override
    public String toString(){
        return "The " + getRank() + " of " +getSuit();
    }
    //methods
    /**
     * Card Compare
     * Comparison by rank then by suit
     * @param o other card to compare this card to
     * @return 1 if greater,0 if the same,-1 if less
     */
    @Override
    public int compareTo(Card o) {
            int r = (this.getRank().ordinal())-(o.getRank().ordinal());
            if (r==0){
                int s = (this.getSuit().ordinal())-(o.getSuit().ordinal());
                if (s==0){
                    return s;
                } else if (s>0){
                    return -1;
                } else {
                    return 1;
                }
            } else if (r>0){
                return 1;
            } else {
                return -1;
            }
        }
    /**
     * Find the difference in ranks between Cards A and B
     * @param A first Card
     * @param B second Card
     * @return the difference in ranks between two cards
     */
    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()));
    }
    /**
     * find the difference in value between the ranks of Cards A and B
     * @param A first Cards
     * @param B second Card
     * @return the difference between the value of the Ranks of two cards
     */
    public static int differenceValue(Card A, Card B){// returns the difference in values between two cards
        return Math.abs(A.rank.getValue()-B.rank.getValue());
    }
    //Comparator classes
    public static class CompareAscending implements Comparator<Card>{
        /**
         * Rank comparison between c1 and c2
         * @param c1 first card
         * @param c2 second rank
         * @return 1 if c1>c2 0 if they are equal -1 if c1<c2
         */
        @Override
        public int compare(Card c1, Card c2) {
            int n = (c1.getRank().ordinal())-(c2.getRank().ordinal());
            int r = 0;
            if(n>0){
                r = -1;
            } else if(n<0){
                r = 1;
            }
            return r;
        }
    }
    public static class CompareSuit implements Comparator<Card>{
        /**
         * Comparison between c1 and c2
         * @param c1 first card
         * @param c2 second card
         * @return 1 if c1>c2 0 if they are equal -1 if c1<c2
         */
        @Override
        public int compare(Card c1, Card c2) {
```

```java
            int n = (c1.getSuit().ordinal())-(c2.getSuit().ordinal());
            int r = 0;
            if(n>0){
                r = -1;
            } else if(n<0){
                r = 1;
            }
            return r;
        }
    }
    /**
     * creates 3 random Cards and compares them with lamdas to the card passed in
     * @param a card to test
     */
    static void selectTest(Card a){
        System.out.println(a.toString());
        ArrayList<Card> cards = new ArrayList<>();
        for (int i = 0; i <3;i++){
            cards.add(randomCard());
        }
        Card.CompareAscending RankObject = new Card.CompareAscending();
        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)));
    }
    //serialsation
    /**
     *  Writes out a card to a file
     * @param card card to serialize
     * @return the fileName of where the card is saved
     */
    public static String write(Card card){
        String fileName = "/" + card.toString() + ".ser";
        try {
            ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(new
               File(fileName)));
            out.writeObject(card);
            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 Card read(String fileName) {
        try {
            ObjectInputStream in = new ObjectInputStream(new FileInputStream(fileName
               ));
            return (Card) in.readObject();
        } catch (IOException i) {
            i.printStackTrace();
            return null;
        } catch (ClassNotFoundException c) {
            System.out.println("Card class not found");
            c.printStackTrace();
            return null;
        }
    }
    /**
     * ADDED
     * For select test to generate random cards
     * Creates a random Card
     * @return a card of random suit and rank
     */
    public static Card randomCard(){
        Random random = new Random();
        Rank[] v = Card.Rank.values();
        return new Card(v[random.nextInt(13)],Suit.getRandom());
    }
}
```

Deck.java

```java
package question1;

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("Deserialization of shuffle");
        deck = read(shuf);
        assert deck != null;
        System.out.println(deck.toString());
        System.out.println("Deserialization 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++) {
            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--];
        }
        /**
         * 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();
            return null;
        }
    }
    /**
     * ADDED
     * ToString
     * @return String of all the cards in the Deck on different lines
     */
    @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();
    }
}
```

Hand.java

```java
package question1;

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
        System.out.println("Count Rank TEN: "+ h.countRank(Card.Rank.TEN));//testing count rank
        //deserializeation
        System.out.println("Original");
        h=read(org);
        assert h != null;
        System.out.println(h.toString());
        System.out.println("AfterRemoval");
        h=read(afterRem);
        assert h != null;
        System.out.println(h.toString());
        System.out.println("\nSorting");//testing sorting
        Hand sort = new Hand();
        for (int i = 0;i<10;i++){
            sort.add(Card.randomCard());
        }
        System.out.println(sort.toString());
        System.out.println("Ascending");
        sort.sortAscending();
        System.out.println(sort.toStringSorted());
        System.out.println("Decending");
        sort.sortDescending();
        System.out.println(sort.toStringSorted());
        System.out.println("Original order=\n" + sort.toString());//shows original order of cards after sorting
    }
    /**
     * Constructor
     * sets hand to an empty arraylist
     */
    public Hand(){
        hand = new ArrayList<>();
    }
    /**
     * Constructor
     * Creates a new hand with the cards in the arraylist
     * @param arrayToAdd arraylist that they hand is initialised to
     */
    public Hand(Card[] arrayToAdd){
        this();
        add(Arrays.asList(arrayToAdd));
    }
    /**
     * Constructor
     * add a hand into the new hand
     * @param handToAdd hand to add into the new hand
     */
    public Hand(Hand handToAdd){
        this();
        add(handToAdd);
    }
    /**
     * Add a card to a hand
     * @param card card to add to hand
     */
    public void add(Card card){
        hand.add(card);
        Card.Rank cardRank = card.getRank();
        rankCount[cardRank.ordinal()]++;
    }
    /**
     * Add a collection to the hand
     * @param collectionToAdd the collection to add to the hand
     */
    public void add(Collection<Card> collectionToAdd){
        for (Card card :collectionToAdd){
            add(card);
        }
    }
    /**
     * Add another hand to the current one
     * @param handToAdd hand to add to the current one
     */
    public void add(Hand handToAdd){
        add(handToAdd.hand);
    }
    /**
     * Removes a card from the hand
     * @param card card to remove
     * @return True if the card was in the hand and False if it wasnt
     */
    public boolean remove(Card card){
        if (hand.contains(card)){
            Card.Rank cardRank = card.getRank();
            rankCount[cardRank.ordinal()]--;
            hand.remove(card);
            return true;
        } else {
            return false;
        }
    }
    /**
     * Removes all the cards int one hand from another hand
     * @param handToRemove hand to remove
     * @return True if all the cards were in the hand and False if it werent
     */
    public boolean remove(Hand handToRemove){
        boolean n = true;
        for (Card card:handToRemove.hand) {
            boolean rem = remove(card);
            if (!rem){
                n=false;
            }
        }
        return n;
    }
    /**
     * removes card in index
     * @param index the index of the card to remove
     * @throws IndexOutOfBoundsException
     */
    public void remove(int index) throws IndexOutOfBoundsException{
        remove(hand.get(index)); //List.get() throws index out of bounds exception if
             out of range
    }
    /**
     * @return the hand iterator
     */
    public Iterator<Card> iterator(){
        return new HandIterator(this);
    }
    /**
     * Iterates through the hand in last added first
     */
    class HandIterator implements Iterator<Card> {
        private int nextCard;
        private int size;
        /**
         *  Initilises next card and size
         * @param h hand to iterate
         */
        public HandIterator(Hand h) {
            this.nextCard = 0;
            this.size = h.hand.size();
        }
        /**
         * @return if the next card is there
         */
        @Override
        public boolean hasNext(){
            return nextCard < size;
        }
        /**
         * gets the next card that was added
         * @return the next card
         */
        @Override
        public Card next(){
            if(!hasNext()) {
                throw new NoSuchElementException();
            }
            return hand.get(nextCard++);
        }
    }
    /**
     *  Writes out a hand to a file
     * @param hand hand to serialize
     * @param file file to write to
     */
    public static void write(Hand hand,String file){
        try {
            ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(new
               File(file)));
            out.writeObject(hand);
            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 Hand read(String fileName) {
        try {
            ObjectInputStream in = new ObjectInputStream(new FileInputStream(fileName
               ));
            return (Hand) in.readObject();
        } catch (IOException i) {
            i.printStackTrace();
            return null;
        } catch (ClassNotFoundException c) {
            System.out.println("Hand class not found");
            c.printStackTrace();
            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();
    }
```

```java
    /**
     * Checks if the hand is a flush
     * @return if the hand is flush True, else False
     */
    public boolean isFlush(){
        Iterator<Card> iterator = this.iterator();
        Card.Suit firstSuit = iterator.next().getSuit();
        while (iterator.hasNext()){
            Card card = iterator.next();
            if (!card.getSuit().equals(firstSuit)){
                return false;
            }
        }
        return true;
    }
    /**
     * Gets if the hand is a straight
     * @return true if all the cards in the hand are consecutive ranks
     */
    public boolean isStraight(){
        sortDescending();
        Iterator<Card> iterator = this.sortIterator();
        Card.Rank Prev = null;
        while (iterator.hasNext()){
            Card card = iterator.next();
            if (Prev != null){
                if (!card.getRank().equals(Prev.getPrevious())){
                    return false;
                }
            }
            Prev = card.getRank();
        }
        return true;
    }
    /**
     * ADDED
     * @return the size of the hand
     */
    public int size(){
        return hand.size();
    }
    /**
     * ADDED
     * the hand in order as a string
     * @return the string with all the cards in the hand in order
     */
    public String toStringSorted(){
        StringBuilder str = new StringBuilder();
        if (sortedHand == null){
            sortAscending();
        }
        Iterator<Card> iterator = this.sortIterator();
        while (iterator.hasNext()){
            Card card = iterator.next();
            str.append("  ").append(card.toString()).append("\n");
        }
        return str.toString();
    }
    /**
     * ADDED
     * @return the sorted hand iterator
     */
    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

```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

```java
package question2;

public class BasicPlayer implements Player {
    private Hand h;
    private Strategy s;
    private CardGame g;
    /**
     * Constructor
     * @param s strategy
     * @param g card game
     */
    public BasicPlayer(Strategy s, CardGame g) {
        h=new Hand();
        setStrategy(s);
        setGame(g);
    }
    /**
     * Add card to hand
     * @param c: Card to add
     */
    @Override
    public void addCard(Card c) {
        h.add(c);
    }
    /**
     * Add hand to current hand
     * @param h: hand to add
     */
    @Override
    public void addHand(Hand h) {
        this.h.add(h);
    }
    /**
     * @return number of cards left in hand
     */
    @Override
    public int cardsLeft() {
        return h.size();
    }
    /**
     * Sets the game
     * @param g: the player should contain a reference to the game it is playing in
     */
    @Override
    public void setGame(CardGame g) {
        this.g = g;
    }
    /**
     * Sets the strategy
     * @param s: the player should contain a reference to its strategy
     */
    @Override
    public void setStrategy(Strategy s) {
        this.s=s;
    }
    /**
     * chooses the bid for the player and then removes the bid played from the hand
     * @param b: the last bid accepted by the game. .
     * @return new bid to be played by the player
     */
    @Override
    public Bid playHand(Bid b) {
        Bid nb = s.chooseBid(b,h,s.cheat(b,h));
        h.remove(nb.getHand());
        return nb;
    }
    /**
     * @param b: the last players bid
     * @return Whether the player calls cheat or not
     */
    @Override
    public boolean callCheat(Bid b) {
        return s.callCheat(h,b);
    }
    //accessors
    /**
     * ADDED
     * @return strategy
     */
    public Strategy getS() {
        return s;
    }
    /**
     * ADDED
     * @return hand
     */
    public Hand getH() {return h;}
    /**
     * ADDED
     * @return game
     */
    public CardGame getG() {return g;}
}
```

BasicCheat.java

```java
package question2;

import java.util.*;
public class BasicCheat implements CardGame{
    private Player[] players;
    private int nosPlayers;
    public static final int MINPLAYERS=3;
    private int currentPlayer;
    private Hand discards;
    private Bid currentBid;
    private boolean notALLComputer = false;
//    static int correctCallsMade = 0;
//    static int incorrectCallsMade = 0;
//    static int callsAgainstCorrect = 0;
//    static int callsAgainstIncorrect = 0;
    // static variable single_instance of type Singleton
    private static BasicCheat singleInstance = null;

    public static void main(String[] args){
        test(1000,3);
    }
    /**
     * my testing function that plays games and prints the percentage that each player won
     */
    public static void test(int numberOfGames,int playerNumber) {
        if (playerNumber<MINPLAYERS){
            playerNumber=MINPLAYERS;
        }
        int[] winners = new int[playerNumber];
        BasicCheat cheat;
        for (int i = 1;i<numberOfGames+1;i++){
            System.out.println("Gamenum: " + i);
            cheat=new BasicCheat(playerNumber);
            int w = cheat.playGame();
            winners[w-1]++;
        }
        for (int i = 0;i<winners.length;i++) {
            if (numberOfGames/100>=1){
                double divider = numberOfGames/100;
                System.out.println((i+1) + ": " +winners[i]/divider + "%");
            } else {
                double multplyer = 100/numberOfGames;
                System.out.println((i+1) + ": " +winners[i]*multplyer + "%");
            }

        }
        //System.out.println("CorrectCalls: " + correctCallsMade + "\nIncorrectCalls: " +incorrectCallsMade + "\nSuccessfulCallsAgainst: " + callsAgainstCorrect+"\nFailtedCallsAgainst: " +callsAgainstIncorrect);
    }
    /**
     * Constructor
     */
    private BasicCheat(){
        this(MINPLAYERS);
    }
    /**
     * Constructor
     * @param n number of players
     */
    private BasicCheat(int n){
        nosPlayers=n;
        players=new Player[nosPlayers];
        for(int i=0;i<nosPlayers;i++)
            players[i]=(new BasicPlayer(new BasicStrategy(),this));
        currentBid=new Bid();
        Card.Rank[] v = Card.Rank.values();
        Random random = new Random();
        currentBid.setRank(v[random.nextInt(13)]);
        currentPlayer=0;
        singleInstance = this;
    }
    /**
     * ADDED
     * Gets the player
     * @param i index of player
     * @return Basicplayer of the player at index i
     */
    public BasicPlayer getPlayer(int i){
        return (BasicPlayer) players[i];
    }
    /**
     * ADDED
     * @return current player
     */
    public int getCurrentPlayer(){
        return currentPlayer;
    }
    /**
     * Plays the turn of the current player
     * @return true
     */
    @Override
    public boolean playTurn(){
        //lastBid=currentBid;
        //Ask player for a play,
        System.out.println("current bid = "+currentBid);
        currentBid=players[currentPlayer].playHand(currentBid);
        System.out.println("Player bid = "+currentBid);
        //Add hand played to discard pile
        discards.add(currentBid.getHand());
        //Offer all other players the chance to call cheat
        boolean cheat=false;
        for(int i=0;i<players.length && !cheat;i++){
            if(i!=currentPlayer){
                cheat=players[i].callCheat(currentBid);
                if(cheat){
                    System.out.println("Player called cheat by Player "+(i+1));
                    if (isCheat(currentBid)){
                        //CHEAT CALLED CORRECTLY
                        players[currentPlayer].addHand(discards);
                        System.out.println("Player " + (currentPlayer+1) + " cheats!"
                           );
//                        if (i==1){
//                            correctCallsMade+=1;
//                        }
//                        if (currentPlayer==1){
//                            callsAgainstCorrect +=1;
//                        }
                    } else {
                        //CHEAT CALLED INCORRECTLY
//                        if (i==1){
//                            incorrectCallsMade+=1;
//                          }
//                          if (currentPlayer==1){
//                              callsAgainstIncorrect +=1;
//                          }
                        System.out.println("Player " + (currentPlayer+ 1) + " Honest");
                        currentPlayer=i;
                        players[currentPlayer].addHand(discards);
                    }
                    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
                    currentBid=new Bid();
                    Card.Rank[] v = Card.Rank.values();
                    Random random = new Random();
                    currentBid.setRank(v[random.nextInt(13)]);
                    //Discards now reset to empty
                    discards=new Hand();
                    for(Player play: players){
                        BasicPlayer p = (BasicPlayer) play;
                        if (p.getS() instanceof ThinkerStrategy){
                            ThinkerStrategy T = (ThinkerStrategy) p.getS();
                            T.cheatCalled();
                        } else if (p.getS() instanceof MyStrategy){
                            MyStrategy M = (MyStrategy) p.getS();
                            M.cheatCalled(currentPlayer);
                        }
                    }
                }
            }
        }
        if(!cheat){
        //Go to the next player
            System.out.println("No Cheat Called");
            currentPlayer=(currentPlayer+1)%nosPlayers;
        }
        return true;
    }
    /**
     * Determines if the game has been won by anyone yet
     * @return -1, if nobody has won or player index if they have
     */
    public int winner(){
        for(int i=0;i<nosPlayers;i++){
            if(players[i].cardsLeft()==0)
                return i;
        }
        return -1;
    }
    /**
     * Initialises the game with PLayer 1 as Thinker and Player 2 as My
     */
    public void initialise(){
        StrategyFactory sf = new StrategyFactory();
        players[0].setStrategy(sf.factory("Thinker"));
        players[1].setStrategy(sf.factory("My"));
        //Create Deck of cards
        Deck d=new Deck();
        d.shuffle();
        //Deal cards to players
        Iterator<Card> it=d.iterator();
        int count=0;
        while(it.hasNext()){
```

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


    }
    /**
     * Actually plays the game
     * @return the winner
     */
    public int playGame(){
        initialise();
        int c=0;
        int w = -1;
        Scanner in = new Scanner(System.in);
        boolean finished=false;
        while(!finished){
            //Play a hand
            System.out.println(" Cheat turn for player "+(currentPlayer+1));
            playTurn();
//            System.out.println(" Current discards =\n"+discards);
//            for (Player p:players){
//                BasicPlayer bp = (BasicPlayer) p;
//                bp.getH().sortAscending();
//                System.out.println("New player Hand\n" + bp.getH().toStringSorted()
    );
//            }
            c++;
            System.out.println(" Turn "+c+ " Complete. Press any key to continue or
               enter Q to quit>");
            if(notALLComputer){
                String str=in.nextLine();
                if(str.equals("Q")||str.equals("q")||str.equals("quit"))
                    finished=true;
            }
            w=winner();
            if(w>=0){
                System.out.println("The Winner is Player "+(w+1));
                finished=true;
            }
        }
        return w+1;
    }
    /**
     * Checks if the player is cheating
     * @param b bid of the player
     * @return true if they are cheating, false if the bid isnt cheating
     */
    public static boolean isCheat(Bid b){
        for(Card c:b.getHand()){
            if(c.getRank()!=b.getRank())
                return true;
        }
        return false;
    }
    /**
     * ADDED
     *Gets instance of the BasicCheat
     * @return instance of the basicCheat Game
     */
    public static BasicCheat getInstance(){
        if (singleInstance == null){
            singleInstance = new BasicCheat();
        }
        return singleInstance;
    }
    /**
     * ADDED
     * @return numbere of players
     */
    public int getNosPlayers() {
        return nosPlayers;
    }
}
```

HumanStrategy.java

```
package question2;

import java.util.Iterator;
import java.util.Scanner;

public class HumanStrategy implements Strategy {
    /**
     * @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 If the human has to cheat
     */
    @Override
    public boolean cheat(Bid b, Hand h) {
        if (hasToCheat(b,h)){
            System.out.println("You have to cheat");
            return true;
        } else {
            return false;
        }
    }
    /**
     * ADDED
     * 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.
     * @param h   The players current hand
     * @return if the player has to cheat
     */
    private boolean hasToCheat(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;
    }
    /**
     * gets the bid the human wants to play
     * @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 of the human
     */
    @Override
    public Bid chooseBid(Bid b, Hand h, boolean cheat) {
        Hand bh = new Hand();
        Card.Rank r;
        System.out.println("Your hand is");
        h.sortDescending();
        Iterator<Card> iterator = h.sortIterator();
        int count = 0;
        while (iterator.hasNext()){
            Card card = iterator.next();
            System.out.println("  "+ (h.size() - count) +": " + card.toString());
            count++;
        }
        System.out.println("Last bid was: " + b.getCount() + " " +b.getRank());
        //choosing the cards to play
        System.out.println("Write down the number of the cards you want to play");
        Scanner input = new Scanner(System. in);
        String inputString = input.nextLine();
        String[] cardNumber = inputString.split(",");
        for (String cardNum:cardNumber) {
            if (Integer.parseInt(cardNum) > 0 && Integer.parseInt(cardNum) <=h.size()
               ){
                bh.add(h.getSortedIndex(Integer.parseInt(cardNum) - 1));
            }
        }
        //choosing rank to say
        int number;
        do {
            System.out.println("Choose Rank\n0: " +b.getRank().getPrevious()+"\n1: "
               +b.getRank()+"\n2: " +b.getRank().getNext());
            number = input.nextInt();
        } while (number > 2 || number <0);
        if (number == 0){
            r = b.getRank().getPrevious();
        } else if (number == 1){
            r = b.getRank();
        } else {
            r = b.getRank().getNext();
        }
        h.remove(bh);
        return new Bid(bh,r);
    }
    /**
     * Asks if the human wants to call cheat or not
     * @param h   The players current hand
     * @param b the current bid
     * @return if the human wants to call cheat or not
     */
    @Override
    public boolean callCheat(Hand h, Bid b) {
        h.sortAscending();
        System.out.println("Your hand is\n" + h.toStringSorted());
        System.out.println("Last bid was: " + b.getCount() + " " +b.getRank());
        Scanner input = new Scanner(System. in);
        int number;
        do {
            System.out.println("Do you want to call cheat?\n0:False\n1:True");
            number = input.nextInt();
        } while (number != 0 && number != 1);
        return number != 0;
    }
}
```

ThinkerStrategy.java

```java
package question2;

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

public class ThinkerStrategy implements Strategy {
    private Hand currentKnownDiscards = new Hand();
    private double p;
    private int[] saidRankCount = new int[13];
    /**
     * Constructor
     * @param p probability to call cheat
     */
    public ThinkerStrategy(double p){
        this.p=p;
    }
    /**
     * Sets p to 0.8 as this is a relativly sensible probability
     */
    public ThinkerStrategy(){
        this(0.1);
    }
    /**
     * The Thinker should of course cheat if it has to. It should also occasionally
        cheat when it 'doesnt have to.
     * @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 if its going to cheat
     */
    @Override
    public boolean cheat(Bid b, Hand h) {
        if (hasToCheat(b,h)){
            return true;
        }
        Random random = new Random();
        int i = random.nextInt(5);
        //I took ocassionally to mean 2/5
        return i <= 1;
    }
    /**
     * ADDED
     * Determines is the player has to cheat
     * @param b previous bid
     * @param h hand of player
     * @return if the player has to cheat
     */
    private boolean hasToCheat(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, the Thinker should be more likely to choose higher
     * cards to discard than low cards. If not cheating, it should usually play all
        its
     * cards but occasionally play a random number.
     * @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 that the computer selects
     */
    @Override
    public Bid chooseBid(Bid b, Hand h, boolean cheat) {
        Hand bh = new Hand();
        Card.Rank r;
        if (cheat){
            bh.add(randomCard(h));
            Random random = new Random();
            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().getNext());
            r = b.getRank().getNext();
            if (h.countRank(b.getRank()) > highest){
                highest = h.countRank(b.getRank());
                r = b.getRank();
            }
            if (h.countRank(b.getRank().getPrevious()) > highest){
                r = b.getRank().getPrevious();
            }
            //occasionally (2/5) play a random number not all like below
            Random random = new Random();
            int rand = random.nextInt(5);
            if (rand > 1){
                Iterator<Card> iterator = h.iterator();
                while (iterator.hasNext()){
                    Card card = iterator.next();
                    if (card.getRank() == r){
                        bh.add(card);
                    }
                }
            } else {
                rand = random.nextInt(h.countRank(r)) + 1;
                Iterator<Card> iterator = h.iterator();
                while (iterator.hasNext() && rand > 0){
                    Card card = iterator.next();
                    if (card.getRank() == r){
                        bh.add(card);
                        rand--;
                    }
                }
            }
        }
        currentKnownDiscards.add(bh);
        //add bh cards to said rankcount non essential
        return new Bid(bh,r);
    }
    /**
     * ADDED
     * more likely to choose higher cards to discard than low cards
     * @param h current player hand
     * @return A card that is more likely to be a high rank
     */
    private Card randomCard(Hand h){
        h.sortAscending();
        int min = 0;
        int max = h.size() - 1;
        while (min != max){
            int mid = (max+min)/2;
            Random random = new Random();
            int i = random.nextInt(3);
            if (i == 0){//1/3 chance to move the max down
                max = mid;
            } else {//2/3 to move the min up
                min = mid;
            }
        }
        return h.getSortedIndex(max);
    }

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

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

MyStrategy.java

```
package question2;

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

public class MyStrategy implements Strategy {
    private int own = -1;//own player number
    private int playerNum;//number of players
    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
    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
    /**
     * Constructor
     * Sets up all the variables
     */
    public MyStrategy(){
        BasicCheat x = BasicCheat.getInstance();
        playerNum = x.getNosPlayers();
        knownDiscardRankCount = new int[13];
        saidRankCount = new int[13];
        prevKnownRankCount = new int[playerNum][13];
        currentKnownRankCount = new int[playerNum][13];
    }
    /**
     * If it has to cheat it will
     * 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.
     * @param h   The players current hand
     * @return if the player is going to cheat or not
     */
    @Override
    public boolean cheat(Bid b, Hand h) {
        if (hasToCheat(b,h)){//if it has to cheat
            return true;
        }
        //my algorithm for whether to cheat or not
        Random rand=new Random();
        int rnd = rand.nextInt(101);//random number between 1 and 100
        //giving it a chance to cheat
        return rnd < cheatLessWithLessCardsInHand(h);
    }
    /**
     * ADDED
     * Determies is the computer has to cheat
     * @param b previous bid
     * @param h current hand
     * @return if the computer has to cheat
     */
    private boolean hasToCheat(Bid b, Hand h) {
        Card.Rank br = b.getRank();
        Iterator<Card> iterator = h.iterator();//iterate through all the cards in the
             hand
        while (iterator.hasNext()){
            Card card = iterator.next();
            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
```

```
          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
            count++;//add another to the cards to be played
          }
        }
        bh.add(getFurthestCards(h,count,r));//add the cards to be played to the hand
      } else {
        if (cheat){
          //cheating but not having to
          //System.out.println("Cheating but not having to===");
          r=getRankToPlay(h,b);
          int quota = h.countRank(r);//get the number of cards to pretend to have
          float rnd = rand.nextFloat();
          if(rnd>=0.9&&quota>1){//90% play full number you can pretend to play
            quota--;
          }
          bh.add(getFurthestCards(h,quota,r));//get the cards to be added to hand
        } else {
          //not cheating
          //System.out.println("Not cheating +++");
          r=getRankToPlay(h,b);
          int quota = h.countRank(r);
          float rnd = rand.nextFloat();
          if(rnd>=0.95&&quota>1){//95% of the time add all the cards
            quota--;
          }
          Iterator<Card> iterator = h.iterator();
          while (iterator.hasNext() && quota > 0) {
            Card card = iterator.next();
            if (card.getRank() == r){//iterate through
              bh.add(card);//add the cards of the correct rank
              quota--;
            }
          }
        }
      }
      int[] bidRankCount = getRankCountArray(bh);
      for (int i = 0;i<bidRankCount.length;i++){
        knownDiscardRankCount[i] += bidRankCount[i];//add all the cards to the known discard rank count
      }
      return new Bid(bh,r);
    }
    /**
     * ADDED
     * 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
     * Gets the rank with the most adjacent cards so its easier to call cheat
     * @param h hand of hte player
     * @param b bid of the player
     * @return the rank the player should play
     */
    private Card.Rank getRankToPlay(Hand h, Bid b){
      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
      int currentCount = h.countRank(b.getRank());//gets the current rank count
```

```java
        int nextCount = h.countRank(b.getRank().getNext()); //gets the next rank count
        int nextNextCount = h.countRank(b.getRank().getNext().getNext()); // gets hte next next rank count
        int prevSum = 0;
        int currentSum = 0;
        int nextSum = 0;
        if (prevCount != 0){
            prevSum = prevPrevCount + prevCount + currentCount;//sums the rank counts of all the ranks around previous
        }
        if (currentCount != 0){
            currentSum = prevCount + currentCount + nextCount;//sums all the rank counts of all the ranks around current
        }
        if (nextCount != 0){
            nextSum = currentCount + nextCount + nextNextCount;//sums all the rank counts of all the ranks around the next
        }
        if (nextSum > currentSum && nextSum >= prevSum){
            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){
            return b.getRank();//returns current if current is greater than or equal to nextsum and greater than or equal to next
        } else {
            return b.getRank().getPrevious();//if it is the highest return previous
        }
    }
    /**
     * ADDED
     * Gets the rank to play when has to cheat using saidRankCount
     * @param b last bid
     * @return rank to play
     */
    private Card.Rank getRankToPlayCheating(Bid b){
        Card.Rank br = b.getRank();
        int saidCountPrev = saidRankCount[br.getPrevious().ordinal()];
        int saidCountCurr = saidRankCount[br.ordinal()];
        int saidCountNext = saidRankCount[br.getNext().ordinal()];
        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++){
            if (highestPrev<prevKnownRankCount[p][br.getPrevious().ordinal()]&&p!=own){
                highestPrev = prevKnownRankCount[p][br.getPrevious().ordinal()];
            }
        }
        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++){
            if (highestNext<prevKnownRankCount[p][br.getNext().ordinal()]&&p!=own){
                highestNext = prevKnownRankCount[p][br.getNext().ordinal()];
            }
        }
        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++){
            if (highestCurrent<prevKnownRankCount[p][br.ordinal()]&&p!=own){
                highestCurrent = prevKnownRankCount[p][br.ordinal()];
            }
        }
        if (highestPrev<4&&highestNext<4&&highestCurrent<4){
            //can play all
            if (saidCountPrev<=saidCountCurr&&saidCountPrev<saidCountNext){
                return br.getPrevious();
            }else if(saidCountNext<=saidCountCurr&&saidCountNext<saidCountPrev){
                return br.getNext();
            }else{
                return br;
            }
        } else if (highestPrev==4&&highestNext<4&&highestCurrent<4){
            //cant play prev without someone possibly knowing
            if(saidCountNext<=saidCountCurr){
                return br.getNext();
            }else{
                return br;
            }
        } else if (highestPrev<4&&highestNext==4&&highestCurrent<4){
            //cant play next without somebody possibly knwing
            if(saidCountPrev<=saidCountCurr){
                return br.getPrevious();
            }else{
                return br;
            }
        } else if (highestPrev<4&&highestNext<4&&highestCurrent==4){
            //cant play current without somebody possibly knowing
            if(saidCountNext<saidCountPrev){
                return br.getNext();
            }else{
                return br.getPrevious();
            }
        } else if (highestPrev==4&&highestNext==4&&highestCurrent<4){
            return br;//has to pick else somebpody will possibly know
        } else if (highestPrev<4&&highestNext==4&&highestCurrent==4){
            return br.getPrevious();//has to pick will possibly
        } else if (highestPrev==4&&highestNext<4&&highestCurrent==4){
            return br.getNext();
        } else {
            //cant play any so just pick the lowest said rank trying to get out away from having to cheat
            if(saidCountNext<saidCountPrev){
                return br.getNext();
            }else{
                return br.getPrevious();
            }
        }
    }
    /**
     * ADDED
     * gets the cards furthest away from being played, and tries to cheat them in to improve its chances of having to cheat
     * @param h the hand
     * @param n the number of cards to return
     * @return a hand of cards to play
     */
    private Hand getFurthestCards(Hand h, int n, Card.Rank rankToPlay){
        Hand rh= new Hand();
        Hand dummyHand = new Hand(h);
        Card cardtoPlay = new Card(rankToPlay, Card.Suit.CLUBS);//temp card
        for (int quotaFull=0;quotaFull<n;quotaFull++){
            Iterator<Card> iterator = dummyHand.iterator();
            Card.Rank furthestRank = null;
            int distance = 0;
            while (iterator.hasNext()) {//gets the furthest distance rank from play
                Card card = iterator.next();
                if(Card.difference(card,cardtoPlay)>distance){
                    distance= Card.difference(card,cardtoPlay);
                    furthestRank = card.getRank();
                }
            }
            boolean added = false;
            iterator = dummyHand.iterator();
            while (iterator.hasNext()&&!added) {
                Card card = iterator.next();
                if(card.getRank() == furthestRank){
                    rh.add(card);//adds the furthest card to the hand to play
                    dummyHand.remove(card);
                    added=true;
                }
            }
        }
        return rh;
    }
    /**
     * ADDED
     * @param h hand to count
     * @return the rankcount of the hand
     */
    private int[] getRankCountArray(Hand h){
        int[] rc = new int[13];
        for(int i = 0;i<13;i++){
            rc[i] = h.getRankCount(i);
        }
        return rc;
    }
    /**
     * Calls cheat if it is the players final card
     * 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
     * @param h   The players current hand
     * @param b the current bid
     * @return true if the player wants to call cheat
     */
    @Override
    public boolean callCheat(Hand h, Bid b) {
        saidRankCount[b.getRank().ordinal()] += b.getCount();//puts the bid into said rank count array
        BasicCheat x = BasicCheat.getInstance();
        subtractionOfRankCount(x.getCurrentPlayer(),b.getCount());
        BasicPlayer p = x.getPlayer(x.getCurrentPlayer());
        if (p.cardsLeft()==0){
            return true;//calls on players final turn meaning plays have to tell the truth on thier last turn
        }
        //use the knowndiscards,player hand,currentknowhplayer hands to call cheat
        int total = b.getCount();
        int rOrd = b.getRank().ordinal();
        total += knownDiscardRankCount[rOrd];
        total += h.countRank(b.getRank());
        for (int i = 0;i<playerNum;i++){
            if (i!=x.getCurrentPlayer() && i!=own){
                total += currentKnownRankCount[i][rOrd];
            }
        }
        if (total > 4){
            return true;
        }
        Random rand=new Random();
        float rnd = rand.nextFloat();
        int copyOfSaidRankCount = saidRankCount[b.getRank().ordinal()];
        if (copyOfSaidRankCount > 4){
            while(rnd>=beMoreSureWithMoreCardsInDiscard()&&copyOfSaidRankCount>=4){//
                be more sure on calling cheat
                rnd = rand.nextFloat();
                copyOfSaidRankCount--;
            }
            return copyOfSaidRankCount > 4;
        }
        return false;
    }
    /**
     * ADDED
     * 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
     */
    private double beMoreSureWithMoreCardsInDiscard(){
        double prob;
        int numInDiscard = 0;
        for (int i = 0;i<13;i++){
            numInDiscard+=knownDiscardRankCount[i]+saidRankCount[i];
        }
        prob = Math.cos(numInDiscard * Math.PI/104)/5;//set to the cosine wave I
            designed
        return prob;
    }
    /**
     * ADDED
     * Called when a cheat is called in the game to reset all the known info
     * Called to reset all the known hands and arrays
     * @param p player whos hand the discards were added too
     */
    public void cheatCalled(int p){
        if (p!=own){
            int[] old = prevKnownRankCount[p];
            for (int i=0;i<playerNum;i++){
                if (i!=p){
                    prevKnownRankCount[i]=currentKnownRankCount[i];
                }
            }
            for(int i=0;i<13;i++){
                old[i]+=knownDiscardRankCount[i];
            }
            prevKnownRankCount[p] = old;
        }
        knownDiscardRankCount = new int[13];
        saidRankCount = new int[13];
    }
    /**
     * ADDED
     * Called each turn to subtract the known rank counts, so that the known cards
         are 100%accurate
     * @param playerNumber player who played cards
     * @param cardsPlayed The number of cards played
     */
    private void subtractionOfRankCount(int playerNumber,int cardsPlayed){
        int[] playersRankCount = currentKnownRankCount[playerNumber];
        for (int i = 0;i<playersRankCount.length;i++){
            if (playersRankCount[i]<=cardsPlayed){
                playersRankCount[i]=0;
            }else{
                playersRankCount[i]-=cardsPlayed;
            }
        }
        currentKnownRankCount[playerNumber]=playersRankCount;
    }
}
```

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("Differance 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)];
            }
        }
        /**
```

```java
         * @return returns the integer value of the card
         */
        public int getValue(){
            return value[this.ordinal()];
        }
    }
    /**
     * Suit enum containing all the suits
     */
    public enum Suit {
        CLUBS, DIAMONDS, HEARTS, SPADES;
        private static Suit[] v = values();
        /**
         * @return a randomly selected suit
         */
        static Suit getRandom(){
            Random random = new Random();
            return v[random.nextInt(4)];
        }
    }
    /**
     * Constructor
     * @param rank the rank the card is
     * @param suit the suit the card is
     */
    public Card(Rank rank, Suit suit){
        this.rank=rank;
        this.suit=suit;
    }
    //accessors methods
    /**
     * @return the rank of the card
     */
    public Rank getRank(){
        return rank;
    }
    /**
     * @return the suit of the card
     */
    public Suit getSuit(){
        return suit;
    }
    //to string
    /**
     * toString method of the card
     * @return a string with of rank and suit
     */
    @Override
    public String toString(){
        return "The " + getRank() + " of " +getSuit();
    }
    //methods
    /**
     * Card Compare
     * Comparison by rank then by suit
     * @param o other card to compare this card to
     * @return 1 if greater,0 if the same,-1 if less
     */
    @Override
    public int compareTo(Card o) {
            int r = (this.getRank().ordinal())-(o.getRank().ordinal());
            if (r==0){
                int s = (this.getSuit().ordinal())-(o.getSuit().ordinal());
                if (s==0){
                    return s;
                } else if (s>0){
                    return -1;
                } else {
                    return 1;
                }
            } else if (r>0){
                return 1;
            } else {
                return -1;
            }
        }
    /**
     * Find the difference in ranks between Cards A and B
     * @param A first Card
     * @param B second Card
     * @return the difference in ranks between two cards
     */
    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()));
    }
    /**
     * find the difference in value between the ranks of Cards A and B
     * @param A first Cards
     * @param B second Card
     * @return the difference between the value of the Ranks of two cards
     */
    public static int differenceValue(Card A, Card B){// returns the difference in values between two cards
        return Math.abs(A.rank.getValue()-B.rank.getValue());
    }
    //Comparator classes
    public static class CompareAscending implements Comparator<Card>{
        /**
         * Rank comparison between c1 and c2
         * @param c1 first card
         * @param c2 second rank
         * @return 1 if c1>c2 0 if they are equal -1 if c1<c2
         */
        @Override
        public int compare(Card c1, Card c2) {
            int n = (c1.getRank().ordinal())-(c2.getRank().ordinal());
            int r = 0;
            if(n>0){
                r = -1;
            } else if(n<0){
                r = 1;
            }
            return r;
        }
    }
    public static class CompareSuit implements Comparator<Card>{
        /**
         * Comparison between c1 and c2
         * @param c1 first card
         * @param c2 second card
         * @return 1 if c1>c2 0 if they are equal -1 if c1<c2
         */
        @Override
        public int compare(Card c1, Card c2) {
            int n = (c1.getSuit().ordinal())-(c2.getSuit().ordinal());
            int r = 0;
            if(n>0){
                r = -1;
            } else if(n<0){
                r = 1;
            }
            return r;
        }
    }
    /**
     * creates 3 random Cards and compares them with lamdas to the card passed in
     * @param a card to test
     */
    static void selectTest(Card a){
        System.out.println(a.toString());
        ArrayList<Card> cards = new ArrayList<>();
        for (int i = 0; i <3;i++){
            cards.add(randomCard());
        }
        Card.CompareAscending RankObject = new Card.CompareAscending();
        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)));
    }
    //serialsation
    /**
     *  Writes out a card to a file
     * @param card card to serialize
     * @return the fileName of where the card is saved
     */
    public static String write(Card card){
        String fileName = "/" + card.toString() + ".ser";
        try {
            ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(new
               File(fileName)));
            out.writeObject(card);
            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 Card read(String fileName) {
        try {
            ObjectInputStream in = new ObjectInputStream(new FileInputStream(fileName
               ));
            return (Card) in.readObject();
        } catch (IOException i) {
            i.printStackTrace();
            return null;
        } catch (ClassNotFoundException c) {
            System.out.println("Card class not found");
            c.printStackTrace();
            return null;
        }
    }
    /**
     * ADDED
     * For select test to generate random cards
     * Creates a random Card
     * @return a card of random suit and rank
     */
    public static Card randomCard(){
        Random random = new Random();
        Rank[] v = Card.Rank.values();
        return new Card(v[random.nextInt(13)],Suit.getRandom());
    }
    /**
     * ADDED
     * needed by Collections.remove in hand
     * @param obj comparison obj
     * @return Card
     */
    @Override
    public boolean equals(Object obj) {
        return obj instanceof Card && obj != null && ((Card) obj).getRank() == this.
           getRank() && ((Card) obj).getSuit() == this.getSuit();
    }
}
```

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++) {
            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--];
        }
        /**
         * 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();
```

```java
            return null;
        }
    }
    /**
     * ADDED
     * ToString
     * @return String of all the cards in the Deck on different lines
     */
    @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();
    }
}
```

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
        System.out.println("Count Rank TEN: "+ h.countRank(Card.Rank.TEN));//testing
            count rank
        //deserializeation
        System.out.println("Original");
        h=read(org);
        assert h != null;
        System.out.println(h.toString());
        System.out.println("AfterRemoval");
        h=read(afterRem);
        assert h != null;
        System.out.println(h.toString());
        System.out.println("\nSorting");//testing sorting
        Hand sort = new Hand();
        for (int i = 0;i<10;i++){
            sort.add(Card.randomCard());
        }
        System.out.println(sort.toString());
        System.out.println("Ascending");
        sort.sortAscending();
        System.out.println(sort.toStringSorted());
        System.out.println("Decending");
        sort.sortDescending();
        System.out.println(sort.toStringSorted());
        System.out.println("Original order=\n" + sort.toString());//shows original
            order of cards after sorting
    }
    /**
     * Constructor
     * sets hand to an empty arraylist
     */
    public Hand(){
        hand = new ArrayList<>();
    }
    /**
     * Constructor
     * Creates a new hand with the cards in the arraylist
     * @param arrayToAdd arraylist that they hand is initialised to
     */
    public Hand(Card[] arrayToAdd){
        this();
        add(Arrays.asList(arrayToAdd));
    }
    /**
     * Constructor
     * add a hand into the new hand
     * @param handToAdd hand to add into the new hand
     */
    public Hand(Hand handToAdd){
        this();
        add(handToAdd);
    }
    /**
     * Add a card to a hand
     * @param card card to add to hand
     */
    public void add(Card card){
        hand.add(card);
        Card.Rank cardRank = card.getRank();
        rankCount[cardRank.ordinal()]++;
    }
    /**
```

```java
     * Add a collection to the hand
     * @param collectionToAdd the collection to add to the hand
     */
    public void add(Collection<Card> collectionToAdd){
        for (Card card :collectionToAdd){
            add(card);
        }
    }
    /**
     * Add another hand to the current one
     * @param handToAdd hand to add to the current one
     */
    public void add(Hand handToAdd){
        add(handToAdd.hand);
    }
    /**
     * Removes a card from the hand
     * @param card card to remove
     * @return True if the card was in the hand and False if it wasnt
     */
    public boolean remove(Card card){
        if (hand.contains(card)){
            Card.Rank cardRank = card.getRank();
            rankCount[cardRank.ordinal()]--;
            hand.remove(card);
            return true;
        } else {
            return false;
        }
    }
    /**
     * Removes all the cards int one hand from another hand
     * @param handToRemove hand to remove
     * @return True if all the cards were in the hand and False if it werent
     */
    public boolean remove(Hand handToRemove){
        boolean n = true;
        for (Card card:handToRemove.hand) {
            boolean rem = remove(card);
            if (!rem){
                n=false;
            }
        }
        return n;
    }
    /**
     * removes card in index
     * @param index the index of the card to remove
     * @throws IndexOutOfBoundsException
     */
    public void remove(int index) throws IndexOutOfBoundsException{
        remove(hand.get(index)); //List.get() throws index out of bounds exception if
             out of range
    }
    /**
     * @return the hand iterator
     */
    public Iterator<Card> iterator(){
        return new HandIterator(this);
    }
    /**
     * Iterates through the hand in last added first
     */
    class HandIterator implements Iterator<Card> {
        private int nextCard;
        private int size;
        /**
         *  Initilises next card and size
         * @param h hand to iterate
         */
        public HandIterator(Hand h) {
            this.nextCard = 0;
            this.size = h.hand.size();
        }
        /**
         * @return if the next card is there
         */
        @Override
        public boolean hasNext(){
            return nextCard < size;
        }
        /**
         * gets the next card that was added
         * @return the next card
         */
        @Override
        public Card next(){
            if(!hasNext()) {
                throw new NoSuchElementException();
            }
            return hand.get(nextCard++);
        }
    }
    /**
     *  Writes out a hand to a file
     * @param hand hand to serialize
     * @param file file to write to
     */
    public static void write(Hand hand,String file){
        try {
            ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(new
                File(file)));
            out.writeObject(hand);
            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 Hand read(String fileName) {
        try {
            ObjectInputStream in = new ObjectInputStream(new FileInputStream(fileName
                ));
            return (Hand) in.readObject();
        } catch (IOException i) {
            i.printStackTrace();
            return null;
        } catch (ClassNotFoundException c) {
            System.out.println("Hand class not found");
            c.printStackTrace();
            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();
```

```java
    }
    /**
     * Checks if the hand is a flush
     * @return if the hand is flush True, else False
     */
    public boolean isFlush(){
        Iterator<Card> iterator = this.iterator();
        Card.Suit firstSuit = iterator.next().getSuit();
        while (iterator.hasNext()){
            Card card = iterator.next();
            if (!card.getSuit().equals(firstSuit)){
                return false;
            }
        }
        return true;
    }
    /**
     * Gets if the hand is a straight
     * @return true if all the cards in the hand are consecutive ranks
     */
    public boolean isStraight(){
        sortDescending();
        Iterator<Card> iterator = this.sortIterator();
        Card.Rank Prev = null;
        while (iterator.hasNext()){
            Card card = iterator.next();
            if (Prev != null){
                if (!card.getRank().equals(Prev.getPrevious())){
                    return false;
                }
            }
            Prev = card.getRank();
        }
        return true;
    }
    /**
     * ADDED
     * @return the size of the hand
     */
    public int size(){
        return hand.size();
    }
    /**
     * ADDED
     * the hand in order as a string
     * @return the string with all the cards in the hand in order
     */
    public String toStringSorted(){
        StringBuilder str = new StringBuilder();
        if (sortedHand == null){
            sortAscending();
        }
        Iterator<Card> iterator = this.sortIterator();
        while (iterator.hasNext()){
            Card card = iterator.next();
            str.append("  ").append(card.toString()).append("\n");
        }
        return str.toString();
    }
    /**
     * ADDED
     * gets the card in specific index in the hand
     * @param i index
     * @return the card at index
     */
    public Card getIndex(int i){
        return this.hand.get(i);
    }
    /**
     * ADDED
     * if cards havent been sorted yet, defaults to sort Asc
     * @param i index
     * @return card at index of sorted hand
     */
    public Card getSortedIndex(int i){
        if(sortedHand.size() == 0){
            sortAscending();
        }
        return this.sortedHand.get(i);
    }
    /**
     * ADDED
     * @return the sorted hand iterator
     */
    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--);
        }
    }
    /**
     * @param i index of rank
     * @return the count of the rank at index i
     */
    public int getRankCount(int i) {
        return rankCount[i];
    }
}
```