Client.py

```
import java.util.Scanner;
//I affirm that I have carried out the attached academic endeavors with full academic honesty,
in
//accordance with the Union College Honor Code and the course syllabus.
// author: Trevor Atkins
public class Client {
   static final int MIN_SCORE = 0;
   public static void main(String[] args) {
//       Card someCard;
//       someCard = new Card(11, "Hearts");
//       System.out.println(someCard);
//       Deck someDeck;
//       someDeck = new Deck();
//       System.out.println(someDeck);
//       someDeck.shuffle();
//       System.out.println(someDeck);
//       Card card1 = someDeck.dealCard();
//       System.out.println(card1);
//       Card card2 = someDeck.dealCard();
//       System.out.println(card2);
//       someDeck.shuffle();
//       System.out.println(someDeck);
         int score= MIN_SCORE;
         Deck deck= new Deck();
         deck.shuffle();
         CommunityCardSet communityCards= new CommunityCardSet();
         while (!communityCards.isfull()){
            communityCards.addCard(deck.dealCard());
         }
         while (!deck.isEmpty()){
            StudPokerHand hand1= new StudPokerHand(communityCards);
            StudPokerHand hand2= new StudPokerHand(communityCards);
            while(!hand1.full() && !hand2.full() ){
               hand1.addCard(deck.dealCard());
               hand2.addCard(deck.dealCard());
            }
            System.out.println("hand 1 is:"+hand1);
            System.out.println("hand 2 is:"+hand2);
            System.out.println("and the community cards are:"+ communityCards);
            System.out.println("Which is worth more? Or are the they same value? 1 if hand 1, -1
if hand 2, or 0 if tie!");
            Scanner scanner= new Scanner(System.in);
            int userinput = scanner.nextInt();
            if (userinput == hand1.compareTo(hand2)){
               score++;
               System.out.println("You're right, keep going!");
            }
            if (userinput!=hand1.compareTo(hand2)){
               System.out.println("Better luck next time! your score is "+ score);
               break;
            }
```

```
            if (deck.isEmpty()){
                System.out.println("Congrats! You win! Score: "+ score);
            }
        }
    }

}
```

Card.py

```
/**
 *  models a single playing card
 */
public class Card {
    public static final String[] SUITS = {"Spades", "Hearts", "Clubs", "Diamonds"};
    public static final Integer[] RANKS = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14};
    private int[] ranks;
    private String[] suits;
    /**
     * constructor
     *
     * @param rank integer between 2-14
     * @param suit string of each suit (Spades, Hearts, Clubs, Diamonds)
     */
    public Card(Integer rank, String suit) {

        ranks = new int[1];
        ranks[0] = rank;
        suits = new String[1];
        suits[0] = suit;

    }

        /**
         * getter
         * @return rank int from 2-14
         */
    public Integer getRank() {
        return ranks[0];
    }

    /**
     * getter
     * @return suit int from 0-3 (0=Spades, 1=Hearts, 2=Clubs, 3=Diamonds)
     */
    public String getSuit() {
        return suits[0];
    }

    /**
     * get rank as string
     * @return rank as 2-10 or Jack, Queen, King, or Ace
     */
    private String getRankString(){
        int cardRank = this.getRank();
        if (cardRank==11) {
            return "Jack";
        }
        else if (cardRank==12) {
            return "Queen";
```

```
        }
        else if (cardRank==13) {
            return "King";
        }
        else if (cardRank==14) {
            return "Ace";
        }
        else {
            String rankAsString = "" + cardRank;
            return rankAsString;
        }
    }

    /**
     * return card in string format
     * @return card as string
     */
    public String toString()
    {
        return this.getRankString() + " of " + this.getSuit();
    }
}
```

Deck.py

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;
import java.util.Collections;
import java.util.concurrent.ThreadLocalRandom;


/**
 *  models a deck of playing cards
 */
public class Deck {
    private static final String[] SUITS = {"Spades", "Hearts", "Clubs", "Diamonds"};
    private static final Integer[] RANKS = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14};
    private static final int FIRST_INDEX = 0;
    private static final int LAST_INDEX = 51;
    /**
     * constructor
     * creates a deck of cards for each rank and suit
     *
     * @param ArrayList Card
     * @param Integer nextToDeal
     */
    private ArrayList<Card> cards;
    private int nextToDeal;

    public Deck() {
        cards = new ArrayList<>();
        this.nextToDeal = FIRST_INDEX;
        for (Integer rank : RANKS) {
            for (String suit : SUITS) {
                this.cards.add(new Card(rank, suit));
            }
        }

    }


    /**
     * Prints the deck as a string.
     *
     * @return string form of the deck.
     */
    public String toString() {
        StringBuilder sb = new StringBuilder();
        List<Card> undealtDeck = cards.subList(nextToDeal, this.size());
        for (Card convertString : undealtDeck){
            sb.append(" | ");
            sb.append(convertString);
            sb.append(" | ");
```

```
        sb.append("\n");
    }
return sb.toString();
}

/**
 * shuffles the deck by randomly swapping
 * the location of each card's index.
 */
public void shuffle() {

    List changingDeck = cards.subList(nextToDeal, LAST_INDEX);
    int changingDeckSize = changingDeck.size();
    for (int i = 0; i < changingDeckSize; i++) {
        int randCard = ThreadLocalRandom.current().nextInt(nextToDeal, LAST_INDEX - 1);
        Card swapper = cards.get(i);
        cards.set(i, cards.get(randCard));
        cards.set(randCard, swapper);
    }
}

/**
 * Checks if there are any cards left in the deck
 *
 * @return true if deck is empty, false if it is not.
 */
public boolean isEmpty() {
    if (this.cards.size() - this.nextToDeal == FIRST_INDEX) {
        return true;
    }
    return false;
}

public int size() {
    return cards.size();
}

/**
 * Access the card at nextToDeal
 * and then increase the card to deal by 1.
 * @return the card at the index of nextToDeal. (top of the deck)
 */
public Card dealCard() {

    if (isEmpty()) {
        System.out.println("The deck is empty! Try again.");
        return null;
    } else {
            int cardToDeal = nextToDeal;
            nextToDeal++;
            return cards.get(cardToDeal);
        }

    }
```

```
/**
 * Gathers the deck back together.
 */
public void gather(){
    nextToDeal = FIRST_INDEX;
}

}
```

PokerHand.py

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class PokerHand {
    private ArrayList<Card> PokerHand;
    private static final int FIRST_INDEX = 0;
    private static final int HANDLENGTH = 5;
    private static final int UNIDENTIFIED = 9999;
    private static final int FLUSH_VALUE = 5;
    private static final int TWO_PAIR_VALUE = 4;
    private static final int PAIR_VALUE = 3;
    private static final int HIGH_CARD_VALUE = 2;
    private static final int NO_PAIR = 0;
    private static final int PAIR = 2;
    /**
     * Takes in a list of cards to create a Poker hand.
     * @param cardList the list of cards the hand takes in
     */
    public PokerHand(ArrayList<Card> cardList) {
        this.PokerHand = cardList;
    }

    /**
     * Takes a card and adds it to the contents of a Hand
     *
     * @param card Takes in a card
     */
    public void addCard(Card card) {
        if (PokerHand.size() > HANDLENGTH) {
            System.out.println("Hand is full.");
        } else if (PokerHand.size() < HANDLENGTH) {
            PokerHand.add(card);
        }
    }

    /**
     * Gets the i-th card in the hand
     * @param index is the i-th card of the hand
     * @return the i-th card
     */
    public Card get_ith_card(int index){
        if (index < FIRST_INDEX){
            return null;
        }
        else{
            return PokerHand.get(index);
        }
```

```
}

/**
 * Checks to see if the hand is a flush
 * @return true if it is a flush, false
 * if it is not.
 */
public boolean isFlush() {
    boolean flush = true;
        for (int i = FIRST_INDEX; i < HANDLENGTH -1; i++) {
            if (!flush){
                return flush;
            }

            if (PokerHand.get(i).getSuit().equals(PokerHand.get(i + 1).getSuit())) {
                flush = true;
            }

            else{
                flush = false;
            }
        }
    return flush;}

/**
 * Checks to see if the hand has a pair
 * @return true if it has a pair,
 * false if it does not.
 */
public boolean isPair() {
    for (int rank : Card.RANKS) {
        int counter = NO_PAIR;
        for (Card card : this.PokerHand) {
            if (card.getRank() == rank) {
                counter++;
                if (counter == PAIR) {
                    return true;
                }
            }
        }

    }
    return false;
}

/**
 * Gets the rank of the pair or two pair
 * @return the rank of the
 * pair or two pair
 */
private int pairRank(){
    for(int rank: Card.RANKS){
        int counter = NO_PAIR;
        for(Card card: this.PokerHand){
```

```
            if (card.getRank() == rank){
                counter ++;
                if (counter == PAIR){
                    return rank;
                }
            }
        }

    }
    return 0;
}

/**
 * Checks if the two pair is a four of a kind
 * @return true if it is a four of a kind, false if it is not.
 */
private boolean fourkindTwoPair(){
    int counter1 = FIRST_INDEX;
    int counter2 = FIRST_INDEX;
    for (Card card: this.PokerHand){
        if (card.getRank().equals(this.PokerHand.get(FIRST_INDEX).getRank()))
            counter1 ++;
        if (card.getRank().equals(this.PokerHand.get(FIRST_INDEX+1).getRank()))
            counter2++;
    if(counter1 == PAIR + PAIR || counter2 == PAIR + PAIR){
        return true;
    }
    } return false;
}

/**
 * Gets the rank of the four of a kind two pair
 * @return the rank
 */
private int fourkindRank(){
    int counter1 = FIRST_INDEX;
    int counter2 = FIRST_INDEX;
    for(Card card : this.PokerHand){
        if (card.getRank().equals(this.PokerHand.get(FIRST_INDEX).getRank()))
            counter1 ++;
        if (card.getRank().equals(this.PokerHand.get(FIRST_INDEX+1).getRank()))
            counter2 ++;
    if (counter1 == 4)
        return this.PokerHand.get(FIRST_INDEX).getRank();
    if (counter2 == 4)
        return this.PokerHand.get(FIRST_INDEX+1).getRank();
    }
    return fourkindRank();
}

/**
 * Checks to see if the hand has a two pair.
 * @return true if there is a two pair,
 * false if there is not.
```

```
 */
private boolean twoPair(){
    if(this.fourkindTwoPair()){
        return true;
    }
    int skipRank = this.pairRank();
    for(int rank: Card.RANKS){
        int counter = NO_PAIR;
        for(Card card : this.PokerHand){
            if (card.getRank().equals(rank) && card.getRank()!=skipRank){
                counter++;
                if (counter == PAIR){
                    return true;
                }
            }
        }
    }return false;
}

/**
 * Gets the rank of the second pair in a two
 * pair.
 * @return the rank
 */
private int secondPairRank(){
    if (this.fourkindTwoPair()){
        return this.fourkindRank();
    }
    int skipRank = this.pairRank();
    for(int rank: Card.RANKS){
        int counter = NO_PAIR;
        for (Card card: this.PokerHand){
            if (card.getRank().equals(rank) &&card.getRank()!=skipRank){
                counter++;
                if (counter == PAIR){
                    return rank;
                }
            }
        }
    } return 0;
}

/**
 * Creates the label of the type of the hand
 * @return string of the type of hand
 */
private String labelHand(){
    if (this.isFlush())
        return "Flush";

    if (this.twoPair())
        return "Two Pair";

    if (this.isPair())
```

```
        return "Pair";

    else{return "High Card";}
}

/**
 * Turns the type of hand into the value
 * of the hand
 * @return the value of the hand
 */
private int handValue(){
    if (this.labelHand().equals("Flush"))
        return FLUSH_VALUE;
    if (this.labelHand().equals("Two Pair"))
        return TWO_PAIR_VALUE;
    if (this.labelHand().equals("Pair"))
        return PAIR_VALUE;
    if (this.labelHand().equals("High Card"))
        return HIGH_CARD_VALUE;
    return UNIDENTIFIED;
}

/**
 * A list of card ranks in descending order
 * from the Hand
 * @return the list of sorted ranks
 */
private ArrayList<Integer> handValuesSorted(){
    ArrayList<Integer> returnArray = new ArrayList<Integer>();
    for (Card card: this.PokerHand){
        returnArray.add(card.getRank());
    }
    Collections.sort(returnArray);
    Collections.reverse((returnArray));
    return returnArray;
}

/**
 * Creates a hand ready to compare by
 * sorting card ranks in descending order
 * @return the sorted list of card ranks
 */
private ArrayList<Integer> comparableHand(){
    ArrayList<Integer> returnList= new ArrayList<Integer>(this.handValuesSorted());
    //System.out.println(returnList);
    if (this.labelHand().equals("Pair")){
        returnList.remove(((Integer) this.pairRank()));
        returnList.remove((Integer) this.pairRank());
        returnList.add(FIRST_INDEX,this.pairRank());
        returnList.add(FIRST_INDEX, this.pairRank());
        return returnList;
    }
    if (this.labelHand().equals("TwoPair")){
        if (this.pairRank()>this.secondPairRank()){
```

```
                returnList.remove(this.pairRank());
                returnList.remove(this.pairRank());
                returnList.remove(this.secondPairRank());
                returnList.remove(this.secondPairRank());
                returnList.add(FIRST_INDEX, this.secondPairRank());
                returnList.add(FIRST_INDEX, this.secondPairRank());
                returnList.add(FIRST_INDEX, this.pairRank());
                returnList.add(FIRST_INDEX, this.pairRank());
            }
            if(this.pairRank()<this.secondPairRank()){
                returnList.remove(this.pairRank());
                returnList.remove(this.pairRank());
                returnList.remove(this.secondPairRank());
                returnList.remove(this.secondPairRank());
                returnList.add(FIRST_INDEX, this.pairRank());
                returnList.add(FIRST_INDEX, this.pairRank());
                returnList.add(FIRST_INDEX, this.secondPairRank());
                returnList.add(FIRST_INDEX, this.secondPairRank());

            }
        }
        return returnList;}

    /**
     * Checks to see if the hand type
     * is equal to the other
     * @param other the "other" hand
     * @return true if the hand types are equal, false if not.
     */
    private boolean handTypeEqual(PokerHand other)
{return(this.labelHand().equals(other.labelHand())); }


    /** * * *
     Determines how this hand compares to another hand,
     returns positive, negative, or zero depending on the comparison.
     *  @param other The hand to compare this hand to
     * @return a negative number if this is worth LESS than other, zero
     * if they are worth the SAME, and a positive number if this is worth * MORE than other
     */

    public int compareTo(PokerHand other) {
        int index = FIRST_INDEX;

        if (!this.handTypeEqual(other)) {
            if (this.handValue() > other.handValue()) {
                return 1;
            }
            if (this.handValue() < other.handValue()) {
                return -1;
            }
        }
        if (this.handTypeEqual(other)) {
            while (index < HANDLENGTH) {
```

```
            if (this.comparableHand().get(index) > other.comparableHand().get(index)) {
                return 1;
            }
            if (this.comparableHand().get(index) < other.comparableHand().get(index)) {
                return -1;
            }
            index++;
        }

    }return 0;
  }


  /**
   * Pretty prints the hand
   * @return the hand in a string format for
   * readability.
   */
  public String toString(){
      StringBuilder sb = new StringBuilder();
      List<Card> stringHand = PokerHand;
      for (Card convertString : stringHand){
          sb.append(" | ");
          sb.append(convertString);
          sb.append(" | ");
      }
      return sb.toString();
  }
}
```

StudPokerHand.py

```
import java.util.ArrayList;
import java.util.List;

public class StudPokerHand {
   private CommunityCardSet communityCards;
   private ArrayList<Card> holeCards;
   public static final int HANDSIZE = 5;
   public static final int STUDHANDSIZE = 2;
   public static final int FIRST_INDEX = 0;
   public static final int FIRST_HOLECARD = 0;
   public static final int SECOND_HOLECARD = 0;

   /**
    * Constructs a StudPokerHand with the community cards
    *
    * @param communityCards the cards used to create the stud poker hand.
    * @param cardList       the list of cards for hole cards.
    */
   public StudPokerHand(CommunityCardSet communityCards, ArrayList<Card> cardList) {
      this.holeCards = cardList;
      this.communityCards = communityCards;
   }

   /**
    * Constructs a StudPokerHand with the community cards
    *
    * @param communityCards the cards used to create the stud poker hand.
    */
   public StudPokerHand(CommunityCardSet communityCards) {
      this.holeCards = new ArrayList<Card>();
      this.communityCards = communityCards;
   }

   /**
    * Takes in a card and adds it to the hole card list.
    * @param card takes in a card
    */
   public void addCard(Card card) {
      this.holeCards.add(card);
   }

   /**
    * Removes the card from the hand at specified index.
    *
    * @param ithcard the index of the card
    */
```

```
public void removeCard(int ithcard) {
    if (ithcard < FIRST_INDEX) {
        return;
    } else {
        this.holeCards.remove(ithcard);
    }
}

/**
 * Removes the first card in the hand.
 */
public void removeFirstCard() {
    this.removeCard(FIRST_INDEX);
}

/**
 *
 * @return True if the hole cards are full, false if it is not.
 */
public boolean full(){
    return this.holeCards.size()==STUDHANDSIZE;
}

/**
 * Gets every single combination of PokerHands
 * @return the collection of combinations in a list.
 */
private ArrayList<PokerHand> getAll5CardHands() {
    ArrayList<PokerHand> returnHand = new ArrayList<PokerHand>();
    ArrayList<Card> allCards = this.communityCards.getCardSet();
    allCards.add(this.holeCards.get(FIRST_HOLECARD));
    allCards.add(this.holeCards.get(SECOND_HOLECARD));
    ArrayList<ArrayList<Card>> allHands = IdentifyCombo.getCombos(allCards, HANDSIZE);

    for (ArrayList<Card> hands : allHands) {
        PokerHand newHand = new PokerHand(new ArrayList<Card>());
        for (Card card : hands) {
            newHand.addCard(card);
        }
        returnHand.add(newHand);
    }
    return returnHand;

}

/**
 * Finds the best possible combination of five card hand with
 * the Stud Poker Hand and Community Cards.
 * @return the best five card hand
 */
private PokerHand getBest5CardHand(){
    ArrayList<PokerHand>hands=this.getAll5CardHands();
    PokerHand bestSoFar= hands.get(0);
    for(int i=1; i<hands.size(); i++ ){
```

```
            if( hands.get(i).compareTo(bestSoFar)>0){
                bestSoFar= hands.get(i);
            }
        }
        return bestSoFar;
    }

    /**
     * Pretty prints the hole cards.
     * @return String of the cards.
     */
    public String toString(){
        StringBuilder sb = new StringBuilder();
        List<Card> stringHand = this.holeCards;
        for (Card convertString : stringHand){
            sb.append(" | ");
            sb.append(convertString);
            sb.append(" | ");
        }
        return sb.toString();
    }

    /**
     * Determines how this hand compares to another hand, returns
     * positive, negative, or zero depending on the comparison. *
     * @param other The hand to compare this hand to
     * @return a negative number if this is worth LESS than other, zero
     * if they are worth the SAME, and a positive number if this is worth * MORE than other
     */
    public int compareTo(StudPokerHand other){
        return this.getBest5CardHand().compareTo(other.getBest5CardHand());
    }

}
```

IdentifyCombo.py

```
import java.util.ArrayList;
// Had help from a friend for the logic of identifying the combinations
// creates all the combinations of cards given 5 card Poker Hands
public class IdentifyCombo {
    private static final int FIRST_INDEX = 0;
    /**
     *
     * @param inputlist a list of cards
     * @return that list of cards within a list
     */
    public static ArrayList<ArrayList<Card>> listWithListInside (ArrayList<Card> inputlist){
        ArrayList<ArrayList<Card>>returnlist= new ArrayList<ArrayList<Card>>();
        returnlist.add(inputlist);
        return returnlist;

    }
    /**
     * Creates and returns a new list of lists, where each new list is a list
     for the given list with a given prefix added.
     * @param prefix: A single element to prepend onto each list.
     * @param listOfListsToPrepend: The list of the lists to prepend.
     * @return: A list of new lists, containing the contents of each of the
    lists from list_of_lists_to_prepend with the prefix prepended.
     */
    private static ArrayList<ArrayList<Card>> prependToAllLists(Card prefix,
ArrayList<ArrayList<Card>> listOfListsToPrepend){
        for (ArrayList<Card> hand: listOfListsToPrepend){
            hand.add(FIRST_INDEX,prefix);
        }
        return listOfListsToPrepend;

    }
    /**
     * Gets all combinations of a given length chosen from a given list, returns a list of those
combinations.
     * @param chooseFrom: A list of cards to choose from
     * @param targetLength: The target length of the combinations to find
     * @return: A list of lists, where each list is a combination of Card object from choose_from of
length target_len.
    Returns the empty list if choose_from is empty.
     */
    public static ArrayList<ArrayList<Card>> getCombos(ArrayList<Card> chooseFrom, int
targetLength){
```

```
        if (chooseFrom.size()== targetLength){
            return listWithListInside(chooseFrom);
        }
        else if(chooseFrom.size()==0){
            return new ArrayList<ArrayList<Card>>();
        }
        Card prefix= chooseFrom.get(0);
        ArrayList<Card> rest= new ArrayList<Card>();
        for (Card card: chooseFrom.subList(1,chooseFrom.size()-1)){
            rest.add(card);
        }
        ArrayList<ArrayList<Card>>returnlist= new
ArrayList<ArrayList<Card>>(prependToAllLists(prefix, getCombos(rest,targetLength-1)));
        returnlist.addAll(getCombos(rest,targetLength));
        return returnlist;


    }
}
```

CommunityCardSet.py

```
import java.util.ArrayList;
public class CommunityCardSet {
    private ArrayList <Card> CommunityCardSet;
    public static int FIRST_INDEX = 0;

    /**
     * Constructor for the Community Card Set
     */
    public CommunityCardSet(){
        this.CommunityCardSet = new ArrayList<Card>();
    }

    public CommunityCardSet(ArrayList<Card> cardList) {
        this.CommunityCardSet = cardList;
    }

    /**
     * Takes in a card and adds it to the Community Card Set
     * @param card takes in a card
     */
    public void addCard(Card card) {this.CommunityCardSet.add(card);}

    /**
     * Removes the first index or card of the Community Card Set
     */
    public void removeFirstCard() {this.CommunityCardSet.remove(FIRST_INDEX);}

    /**
     *Checks whether the Set is full
     * @return whether true if full, false if it is not.
     */
    public boolean isfull(){
        return this.CommunityCardSet.size()==5;
    }

    /**
     * Pretty prints the Card set as a string for readability.
     * @return the Set as a string.
     */
    public String toString() {return this.CommunityCardSet.toString(); }

    /**
     * Gets the Community Card Set in a list
     * @return the list of cards in the community set.
     */
    public ArrayList<Card> getCardSet(){
        CommunityCardSet commCopy = new CommunityCardSet();
        for(Card card: this.CommunityCardSet){
            commCopy.addCard(card);
        }
        return commCopy.CommunityCardSet;
```

```
    }
}
```

PokerComparisonTests.py

```
import java.util.ArrayList;

public class PokerComparisonTests {
    public static void main(String[] args) {

        final int MIN_SCORE = 0;
//          Testing.startTests();
//          Testing.setVerbose(true);
//          CommunityCardSet communityCards = new CommunityCardSet();
//          Card card1 = new Card(2, "Diamonds");
//          Card card2 = new Card(3, "Diamonds");
//          Card card3 = new Card(4,"Diamonds");
//          Card card4 = new Card(5, "Diamonds");
//          Card card5 = new Card(6, "Diamonds");
//          communityCards.addCard(card1);
//          communityCards.addCard(card2);
//          communityCards.addCard(card3);
//          communityCards.addCard(card4);
//          communityCards.addCard(card5);
//          StudPokerHand hand1 = new StudPokerHand(communityCards);
//          StudPokerHand hand2 = new StudPokerHand(communityCards);
//          Card h1card1 = new Card(8, "Spades");
//          Card h1card2 = new Card(7, "Hearts");
//          Card h2card1 = new Card(9, "Hearts");
//          Card h2card2 = new Card(9, "Clubs");
//          hand1.addCard(h1card1);
//          hand1.addCard(h1card2);
//          hand2.addCard(h2card1);
//          hand2.addCard(h2card2);
//          System.out.printf("\n" + "%d" + "\n" , hand1.compareTo(hand2));
//          Testing.assertEquals("Testing compare to with consecutive ranks and " +
//          "identical suits.", -1, hand1.compareTo(hand2));
//          Testing.finishTests();
        Testing.startTests();
        Testing.setVerbose(true);
        Card card1 = new Card(2, "Diamonds");
        Card card2 = new Card(3, "Diamonds");
        Card card3 = new Card(4,"Diamonds");
        Card card4 = new Card(6, "Diamonds");
        Card card5 = new Card(6, "Hearts");
        Card card6 = new Card(5, "Hearts");
        Card card7 = new Card(5, "Diamonds");
        Card card8 = new Card(8,"Hearts");
        Card card9 = new Card(9, "Hearts");
```

```
        Card card10 = new Card(10, "Hearts");
        PokerHand PokerHand1 = new PokerHand(new ArrayList<Card>());
        PokerHand PokerHand2 = new PokerHand(new ArrayList<Card>());
        PokerHand1.addCard(card1);
        PokerHand1.addCard(card2);
        PokerHand1.addCard(card3);
        PokerHand1.addCard(card4);
        PokerHand1.addCard(card5);
        PokerHand2.addCard(card6);
        PokerHand2.addCard(card7);
        PokerHand2.addCard(card8);
        PokerHand2.addCard(card9);
        PokerHand2.addCard(card10);
        Testing.assertEquals("Testing compare to with hand 1 having a higher rank pair and " +
                "identical suits.", 1, PokerHand1.compareTo(PokerHand2));
        Testing.finishTests();

        Testing.startTests();
        Testing.setVerbose(true);
        Card card_1 = new Card(3, "Diamonds");
        Card card_2 = new Card(4, "Diamonds");
        Card card_3 = new Card(6,"Diamonds");
        Card card_4 = new Card(7, "Diamonds");
        Card card_5 = new Card(8, "Diamonds");
        Card card_6 = new Card(3, "Hearts");
        Card card_7 = new Card(4, "Hearts");
        Card card_8 = new Card(5,"Hearts");
        Card card_9 = new Card(7, "Hearts");
        Card card_10 = new Card(8, "Hearts");
        PokerHand PokerHand01 = new PokerHand(new ArrayList<Card>());
        PokerHand PokerHand02 = new PokerHand(new ArrayList<Card>());
        PokerHand1.addCard(card_1);
        PokerHand1.addCard(card_2);
        PokerHand1.addCard(card_3);
        PokerHand1.addCard(card_4);
        PokerHand1.addCard(card_5);
        PokerHand2.addCard(card_6);
        PokerHand2.addCard(card_7);
        PokerHand2.addCard(card_8);
        PokerHand2.addCard(card_9);
        PokerHand2.addCard(card_10);
        Testing.assertEquals("Testing compare to with flush " +
                "and a different rank for the third highest card.", 1,
PokerHand1.compareTo(PokerHand2));
        Testing.finishTests();

        Testing.startTests();
        Testing.setVerbose(true);
        Card card_01 = new Card(3, "Spades");
        Card card_02 = new Card(3, "Clubs");
        Card card_03 = new Card(6,"Diamonds");
        Card card_04 = new Card(7, "Diamonds");
        Card card_05 = new Card(8, "Diamonds");
        Card card_06 = new Card(3, "Hearts");
```

```
Card card_07 = new Card(13, "Diamonds");
Card card_08 = new Card(13,"Hearts");
Card card_09 = new Card(14, "Clubs");
Card card_010 = new Card(14, "Hearts");
PokerHand PokerHand001 = new PokerHand(new ArrayList<Card>());
PokerHand PokerHand002 = new PokerHand(new ArrayList<Card>());
PokerHand1.addCard(card_01);
PokerHand1.addCard(card_02);
PokerHand1.addCard(card_03);
PokerHand1.addCard(card_04);
PokerHand1.addCard(card_05);
PokerHand2.addCard(card_06);
PokerHand2.addCard(card_07);
PokerHand2.addCard(card_08);
PokerHand2.addCard(card_09);
PokerHand2.addCard(card_010);
Testing.assertEquals("Testing compare to with two pair " +
        "and a pair for the third highest card.", -1, PokerHand1.compareTo(PokerHand2));
Testing.finishTests();

    }
}
```