```java
package proj4;

public class Card {

    private int rank;
    private int suit;

    private final String[] strNumRanks = {"fillerIndex0", "fillerIndex1", "2", "3", "4", "5",
            "6", "7", "8", "9", "10", "11", "12", "13", "14"};
    private final String[] strStringRanks = {"fillerIndex0", "fillerIndex1", "Two", "Three", "Four", "Five",
            "Six", "Seven", "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen"};
    private final String[] strUpperRanks = {"fillerIndex0", "fillerIndex1", "Two", "Three", "Four", "Five",
            "Six", "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King", "Ace"};

    private final String[] strStringSuits = {"Spades", "Hearts", "Clubs", "Diamonds"};

    /**
     * constructor
     * @param rank String: whole cards (2-10) can either be spelled
     * out like "two" or numeric like "2". Case insensitive.
     * @param suit String: "Spades", "Hearts", "Clubs", or "Diamonds"
     */
    public Card(String rank, String suit){

        this.rank = rankString2rankInt(rank);
        this.suit = suitString2suitInt(suit);

    }

```

```
30      /**
31       * constructor
32       * @param rank integer between 2-14
33       * @param suit integer: 0=Spades, 1=Hearts, 2=
   Clubs, or 3=Diamonds
34       */
35      public Card(int rank, int suit){
36
37          this.rank = rank;
38          this.suit = suit;
39
40      }
41
42
43      /**
44       * Converts a String rank to an integer.
45       * @param rank2convert The rank we are converting
    to an integer.
46       * @return The int the rank represents
47       */
48      private int rankString2rankInt(String
   rank2convert){
49
50          for (int index = 2; index < strNumRanks.
   length; index++) {
51              if ((rank2convert.equalsIgnoreCase(
   strNumRanks[index])) ||
52                      (rank2convert.equalsIgnoreCase(
   strStringRanks[index])) ||
53                      (rank2convert.equalsIgnoreCase(
   strUpperRanks[index])))
54                          {
55                          return index;
56                          }
57          }
58          return 999; // will not go off, here for
   syntax purposes
59      }
60
61      /**
62       * Converts a String suit to an integer.
```

```
63      * @param suit2convert The suit we are converting to an integer.
64      * @return The int the suit represents
65      */
66     private int suitString2suitInt(String suit2convert) {
67 
68         for (int index = 0; index < strStringSuits.length; index++){
69             if (suit2convert.equalsIgnoreCase(strStringSuits[index])){
70                 return index;
71             }
72         }
73         return 999; // will not go off, here for syntax purposes
74     }
75 
76     /**
77      * getter for rank
78      * @return card's rank as integer: 2-14
79      */
80     public int getRank() {
81         return rank;
82     }
83 
84 
85 
86     /**
87      * getter for suit
88      * @return string (plural, starts with Capital letter)
89      */
90     public String getSuit() {
91         for (int index = 0; index < strStringSuits.length; index++){
92             if (Integer.toString(suit).equals(Integer.toString(index))){
93                 return strStringSuits[index];
94             }
95         }
```

```
 96          return "Error"; // will not go off, Here for
     syntax purposes
 97      }
 98
 99      /**
100       * Uses getRank() to get an int Rank, then
    converts it back into a String.
101       * @return A stringified version of the int Rank
    .
102       */
103      private String rank2String(){
104          int myRank = getRank();
105          if (myRank == 11) {
106              return "Jack";
107          }
108          else if (myRank == 12) {
109              return "Queen";
110          }
111          else if (myRank == 13) {
112              return "King";
113          }
114          else if (myRank == 14) {
115              return "Ace";
116          }
117          else {  // must be a whole number - just
    convert it to string
118              return "" + myRank;
119          }
120      }
121
122      /**
123       * returns string version of Card
124       * @return string like "Jack of Clubs"
125       */
126      public String toString() {
127          return  rank2String() + " of " + getSuit();
128      }
129
130  }
131
```

```java
package proj4;

import java.util.ArrayList;
import java.util.concurrent.ThreadLocalRandom;

public class Deck {

    private final int[] RANKS = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14};
    private final int[] SUITS = {0,1,2,3};
    private final int MAX_CARDS_IN_DECK = 52;

    private ArrayList<Card> cardDeck;
    private int nextToDeal;

    /**
     *constructor. Makes a deck of 52 playing cards
     */
    public Deck(){

        this.cardDeck = new ArrayList<>(MAX_CARDS_IN_DECK);
        this.nextToDeal = 0;

        for (int rankIndex=0; rankIndex<RANKS.length; rankIndex++){
            for(char suitIndex=0; suitIndex<SUITS.length; suitIndex++){
                cardDeck.add( new Card(RANKS[rankIndex],SUITS[suitIndex]) );
            }
        }
    }

    /**
     * Shuffles the Deck. Only shuffles cards that have NOT been dealt.
     */
    public void shuffle(){
        for (int cardIndex=0; cardIndex<cardDeck.size(); cardIndex++){
```

```
35              int randomIndex = ThreadLocalRandom.
   current().nextInt(nextToDeal,MAX_CARDS_IN_DECK);
36              cardDeck = swap(cardDeck, cardIndex,
   randomIndex);
37
38          }
39      }
40
41      /**
42       * A method to swap all the undealt cards in a
   deck.
43       * @param list an ArrayList of <Card> type</Card>
44       * @param currentIndicy the indicy of the current
    card in our loop
45       * @param randomIndicy the indicy of a random
   undealt card in the deck
46       * @return returns a swapped ArrayList (
   specifically a swapped deck)
47       */
48      private ArrayList<Card> swap(ArrayList<Card> list
   , int currentIndicy, int randomIndicy){
49
50          Card temp1;
51          Card temp2;
52
53          temp1 = list.get(currentIndicy);
54          temp2 = list.get(randomIndicy);
55
56          list.set(currentIndicy,temp2);
57          list.set(randomIndicy, temp1);
58
59          return list;
60      }
61
62      /**
63       * Deals a card, or returns nothing if the deck
   is empty.
64       * @return Returns a card, or null the deck is
   empty.
65       */
66      public Card deal(){
```

```
 67          nextToDeal++;
 68          if(cardDeck.size() > 0) {
 69              return cardDeck.get(nextToDeal - 1);
 70          }
 71          else{
 72              return null;
 73          }
 74      }
 75
 76      /**
 77       * Determines if there are cards left to deal in
     the deck.
 78       * @return true or false, depending on if the
    deck has cards left to deal.
 79       */
 80      public boolean isEmpty() {
 81          if ((MAX_CARDS_IN_DECK - nextToDeal) > 0) {
 82              return false;
 83          } else {
 84              return true;
 85          }
 86      }
 87
 88      /**
 89       * Determines how many cards in the deck have
    not been dealt yet.
 90       * @return integer number of cards not yet dealt
    .
 91       */
 92      public int size(){
 93          return (MAX_CARDS_IN_DECK - nextToDeal);
 94      }
 95
 96      /**
 97       * Returns the deck to a state where no cards
    have been dealt
 98       */
 99      public void gather(){
100          nextToDeal = 0;
101      }
102
```

```
103     /**
104      * Returns all undealt cards remaining in the
    deck
105      * @return An easily readable string of all the
    undealt cards in the deck.
106      */
107     public String toString(){
108 
109         ArrayList<Card> undealtList = new ArrayList
    <>();
110 
111         for (int card=nextToDeal; card<cardDeck.size
    (); card++){
112             undealtList.add( cardDeck.get(card) );
113         }
114 
115         return undealtList.toString();
116     }
117 
118 }
119 
```

```java
package proj4;

/*
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.
 */

import java.util.ArrayList;
import java.util.Scanner;

public class Client {

    public static void main(String[] args) {

        final int MAX_CARDS_IN_HAND = 5;
        final int MAX_CARDS_IN_COMSET = 5;
        final int MAX_CARDS_IN_STUD = 2;
        int points = 0;
        String messageLoss = "Incorrect. Final points: "; // +  points;
        String messageWin = "You win! Final points: "; //+ points;

        Deck newDeck = new Deck();
        newDeck.shuffle(); // create and shuffle deck

        ArrayList<Card> communityArray = new ArrayList<>(); // create community cards
        CommunityCardSet cardSet = new CommunityCardSet(communityArray);

        ArrayList<Card> h1 = new ArrayList<>(); // create studs
        StudPokerHand stud1 = new StudPokerHand(cardSet, h1);
        ArrayList<Card> h2 = new ArrayList<>();
        StudPokerHand stud2 = new StudPokerHand(cardSet, h2);

```

```
33          for (int comToDeal = 0; comToDeal < MAX_CARDS_IN_COMSET; comToDeal++){ // community dealing loop
34              Card card = newDeck.deal();
35              cardSet.addCard(card);
36          }
37
38          while (newDeck.size() > (MAX_CARDS_IN_STUD * 2)){
39
40              for (int cardToDeal = 0; cardToDeal < MAX_CARDS_IN_STUD; cardToDeal++){ // stud dealing loop
41                  Card card1 = newDeck.deal();
42                  stud1.addCard(card1);
43
44                  Card card2 = newDeck.deal();
45                  stud2.addCard(card2);
46              }
47
48              int answer = stud1.compareTo(stud2);
49
50              System.out.println();
51              System.out.println("Which of the two studs can create a more valuable hand?");
52              System.out.println();
53              System.out.println("Community Cards: " + cardSet);
54              System.out.println("Stud 1: " + stud1); // ask teach about redundant printing
55 //              System.out.println();
56              System.out.println("Stud 2: " + stud2); //
57              System.out.println();
58              System.out.println("Answer \"1\" for Stud 1, or \"-1\" for Stud 2, or \"0\" if there is a tie.");
59              Scanner scan = new Scanner(System.in);
60              int guess = scan.nextInt();
61
62              if (answer == guess) {
```

```
63                  points++;
64                  System.out.println("Correct!");
65              } else {
66                  System.out.println(messageLoss +
    points);
67                  System.out.println("The answer was
    : " + answer);
68                  return;
69              }
70
71              h1.clear();
72              h2.clear();
73          }
74
75          System.out.println(messageWin + points);
76
77      }
78  }
79
```

```java
package proj4;

/**
 * This class contains a collection of methods that help with testing.  All methods
 * here are static so there's no need to construct a Testing object.  Just call them
 * with the class name like so:
 * <p></p>
 * <code>Testing.assertEquals("test description", expected, actual)</code>
 *
 * @author Kristina Striegnitz, Aaron Cass, Chris Fernandes
 * @version 5/28/18
 */
public class Testing {

    private static boolean VERBOSE = true;
    private static int numTests;
    private static int numFails;

    /**
     * Toggles between a lot of output and little output.
     *
     * @param verbose
     *            If verbose is true, then complete information is printed,
     *            whether the tests passes or fails. If verbose is false, only
     *            failures are printed.
     */
    public static void setVerbose(boolean verbose)
    {
        VERBOSE = verbose;
    }

    /**
     * Each of the assertEquals methods tests whether the actual
```

```
34      * result equals the expected result. If it does
   , then the test
35      * passes, otherwise it fails.
36      *
37      * The only difference between these methods is
   the types of the
38      * parameters.
39      *
40      * All take a String message and two values of
   some other type to
41      * compare:
42      *
43      * @param message
44      *            a message or description of the
   test
45      * @param expected
46      *            the correct, or expected, value
47      * @param actual
48      *            the actual value
49      */
50     public static void assertEquals(String message,
   boolean expected,
51                                     boolean actual)
52     {
53         printTestCaseInfo(message, "" + expected, ""
    + actual);
54         if (expected == actual) {
55             pass();
56         } else {
57             fail(message);
58         }
59     }
60
61     public static void assertEquals(String message,
   int expected, int actual)
62     {
63         printTestCaseInfo(message, "" + expected, ""
    + actual);
64         if (expected == actual) {
65             pass();
66         } else {
```

```
 67              fail(message);
 68          }
 69      }
 70
 71      public static void assertEquals(String message,
    Object expected,
 72                                      Object actual)
 73      {
 74          String expectedString = "<<null>>";
 75          String actualString = "<<null>>";
 76          if (expected != null) {
 77              expectedString = expected.toString();
 78          }
 79          if (actual != null) {
 80              actualString = actual.toString();
 81          }
 82          printTestCaseInfo(message, expectedString,
    actualString);
 83
 84          if (expected == null) {
 85              if (actual == null) {
 86                  pass();
 87              } else {
 88                  fail(message);
 89              }
 90          } else if (expected.equals(actual)) {
 91              pass();
 92          } else {
 93              fail(message);
 94          }
 95      }
 96
 97      /**
 98       * Asserts that a given boolean must be true.
    The test fails if
 99       * the boolean is not true.
100       *
101       * @param message The test message
102       * @param actual The boolean value asserted to
    be true.
103       */
```

```
104     public static void assertTrue(String message,
    boolean actual)
105     {
106         assertEquals(message, true, actual);
107     }
108 
109     /**
110      * Asserts that a given boolean must be false.
    The test fails if
111      * the boolean is not false (i.e. if it is true
    ).
112      *
113      * @param message The test message
114      * @param actual The boolean value asserted to
    be false.
115      */
116     public static void assertFalse(String message,
    boolean actual)
117     {
118         assertEquals(message, false, actual);
119     }
120 
121     private static void printTestCaseInfo(String
    message, String expected,
122                                           String
    actual)
123     {
124         if (VERBOSE) {
125             System.out.println(message + ":");
126             System.out.println("expected: " +
    expected);
127             System.out.println("actual:   " + actual
    );
128         }
129     }
130 
131     private static void pass()
132     {
133         numTests++;
134 
135         if (VERBOSE) {
```

```
136             System.out.println("--PASS--");
137             System.out.println();
138         }
139     }
140 
141     private static void fail(String description)
142     {
143         numTests++;
144         numFails++;
145 
146         if (!VERBOSE) {
147             System.out.print(description + "  ");
148         }
149         System.out.println("--FAIL--");
150         System.out.println();
151     }
152 
153     /**
154      * Prints a header for a section of tests.
155      *
156      * @param sectionTitle The header that should be
     printed.
157      */
158     public static void testSection(String
    sectionTitle)
159     {
160         if (VERBOSE) {
161             int dashCount = sectionTitle.length();
162             System.out.println(sectionTitle);
163             for (int i = 0; i < dashCount; i++) {
164                 System.out.print("-");
165             }
166             System.out.println();
167             System.out.println();
168         }
169     }
170 
171     /**
172      * Initializes the test suite. Should be called
    before running any
173      * tests, so that passes and fails are correctly
```

```
173  tallied.
174       */
175     public static void startTests()
176     {
177         System.out.println("Starting Tests");
178         System.out.println();
179         numTests = 0;
180         numFails = 0;
181     }
182 
183     /**
184      * Prints out summary data at end of tests.
    Should be called
185      * after all the tests have run.
186      */
187     public static void finishTests()
188     {
189         System.out.println("===============");
190         System.out.println("Tests Complete");
191         System.out.println("===============");
192         int numPasses = numTests - numFails;
193 
194         System.out.print(numPasses + "/" + numTests
     + " PASS ");
195         System.out.printf("(pass rate: %.1f%s)\n",
196                 100 * ((double) numPasses) /
    numTests,
197                 "%");
198 
199         System.out.print(numFails + "/" + numTests
     + " FAIL ");
200         System.out.printf("(fail rate: %.1f%s)\n",
201                 100 * ((double) numFails) / numTests
    ,
202                 "%");
203     }
204 
205 }
206 
```

```java
package proj4;

import java.util.ArrayList;

public class PokerHand {

    private final int[] RANKS = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14};
    private final int HIGHEST_RANK = 14;
    private final int MAX_CARDS_IN_HAND = 5;
    private ArrayList<Card> cardList;

    /**
     * Constructor. models a hand of 5 playing cards.
     * @param cardList A list of cards that should be in the hand
     */
    public PokerHand(ArrayList<Card> cardList){
        this.cardList = cardList;
    }

    /**
     * Adds a card to the hand.
     * If the hand is full, does nothing.
     * @param CardObject The card that will be added.
     */
    public void addCard(Card CardObject){

        if (cardList.size() <= MAX_CARDS_IN_HAND){
            cardList.add(CardObject);
        }
        else {
            return;
        }

    }

    /**
     * Returns a card from the hand.
     * @param index The index of the card we are getting.
```

```
39      * @return returns the card at the sepcified
   index.
40      * If the index is invalid, returns nothing at
   all.
41      */
42     public Card get_ith_card(int index){
43 
44         if ((index >= 0) || (index <
   MAX_CARDS_IN_HAND)){
45             return cardList.get(index);
46         }
47         else {
48             return null;
49         }
50     }
51 
52     /**
53      * Returns all the cards in a hand.
54      * @return Returns an easily readable string of
   all the cards in a hand.
55      */
56     public String toString(){
57 
58         ArrayList<Card> handList = new ArrayList<>();
59         for (int card=0; card<cardList.size(); card
   ++){
60             handList.add( cardList.get(card) );
61         }
62         return handList.toString();
63     }
64 
65     /**
66      * Determines how this hand compares to another
   hand, returns
67      * positive, negative, or zero depending on the
   comparison.
68      *
69      * @param other The hand to compare this hand to
70      * @return a negative number if this is worth
   LESS than other, zero
71      * if they are worth the SAME, and a positive
```

```
 71 number if this is worth
 72      * MORE than other
 73      */
 74     public int compareTo(PokerHand other){
 75 
 76         PokerHand hand1 = PokerHand.this;
 77         PokerHand hand2 = other;
 78 
 79         // hand1 is greater
 80         if (hand1.determineIfFlush() && !hand2.
    determineIfFlush()){
 81             return 1;
 82         }
 83         if (hand1.determineIfTwoPair() &&
 84                 (!hand2.determineIfFlush() && !hand2
    .determineIfTwoPair())){
 85             return 1;
 86         }
 87         if (hand1.determineIfPair() &&
 88                 (!hand2.determineIfFlush() && !hand2
    .determineIfTwoPair() && !hand2.determineIfPair())){
 89             return 1;
 90         }
 91 
 92         // hand2 is greater
 93         if (hand2.determineIfFlush() &&
 94                 !hand1.determineIfFlush()){
 95             return -1;
 96         }
 97         if (hand2.determineIfTwoPair() &&
 98                 (!hand1.determineIfFlush() && !hand1
    .determineIfTwoPair())){
 99             return -1;
100         }
101         if (hand2.determineIfPair() &&
102                 (!hand1.determineIfFlush() && !hand1
    .determineIfTwoPair() && !hand1.determineIfPair())){
103             return -1;
104         }
105 
106         // both hands are the same type
```

```
107         if ((hand1.determineIfFlush() && hand2.
    determineIfFlush()) || // flushes and HighCards
108                 ((hand1.determineIfHighCard() &&
    hand2.determineIfHighCard()) &&
109                         (!hand1.determineIfPair
    () && !hand2.determineIfPair()) &&
110                         (!hand1.determineIfTwoPair
    () && !hand2.determineIfTwoPair()))){
111
112             ArrayList<Integer> h1 = hand1.rankFlush
    ();
113             ArrayList<Integer> h2 = hand2.rankFlush
    ();
114
115             // systemiatically ranking flushes and
    highCards is the same
116
117             for (int card = 0; card<h1.size(); card
    ++){ // iterates through hands
118
119                 if (h1.get(card) > h2.get(card)){
120                     return 1;
121                 }
122                 else if (h1.get(card) < h2.get(card
    )){
123                     return -1;
124                 }
125             }
126
127             return 0;
128
129         }
130
131         if (hand1.determineIfTwoPair() && hand2.
    determineIfTwoPair()){ // twoPairs
132             ArrayList<Integer> h1 = hand1.
    rankTwoPair();
133             ArrayList<Integer> h2 = hand2.
    rankTwoPair();
134
135             for (int card = 0; card<h1.size(); card
```

```
135 ++){ // iterates through hands
136                 if (h1.get(card) > h2.get(card)){
137                     return 1;
138                 }
139                 else if (h1.get(card) < h2.get(card
    )){
140                     return -1;
141                 }
142             }
143 
144             return 0;
145 
146         }
147 
148         if (hand1.determineIfPair() && hand2.
    determineIfPair()){ // pairs
149             ArrayList<Integer> h1 = hand1.rankPair
    ();
150             ArrayList<Integer> h2 = hand2.rankPair
    ();
151 
152             for (int card = 0; card<h1.size(); card
    ++){ // iterates through hands
153                 if (h1.get(card) > h2.get(card)){
154                     return 1;
155                 }
156                 else if (h1.get(card) < h2.get(card
    )){
157                     return -1;
158                 }
159             }
160 
161             return 0;
162 
163         }
164 
165         return 999; // will not go off, here for
    syntax purposes
166     }
167 
168     /**
```

```
169      * Takes the hand and extracts the ranks of all
    cards' ranks
170      * @return Returns an ArrayList of integers in
    easily readable form.
171      */
172     private ArrayList<Integer> ranksOfHand(){
173 
174         ArrayList<Integer> rankList = new ArrayList
    <>();
175         Card cardIndex;
176         int card;
177 
178         for (int i = 0; i<MAX_CARDS_IN_HAND; i++){
179             cardIndex = this.get_ith_card(i);
180             card = cardIndex.getRank();
181             rankList.add(card);
182         }
183         return rankList;
184     }
185 
186     /**
187      * Identifies a hand as a flush.
188      * @return true or false, depending on if hand
    is a flush.
189      */
190     private boolean determineIfFlush(){
191 
192         Card card;
193         String cardSuit;
194 
195         card = this.get_ith_card(0);
196         cardSuit = card.getSuit(); // cardSuit; the
    suit of the first card in a hand
197         for (int cardIndex=0; cardIndex<
    MAX_CARDS_IN_HAND; cardIndex++){ //iterates through
    a hand
198             if (!cardSuit.equalsIgnoreCase(this.
    get_ith_card(cardIndex).getSuit())){
199                 return false;
200             }
201         }
```

```
202             return true;
203         }
204 
205         /**
206          * Identifies a hand as a pair.
207          * @return true or false, depending on if hand
    is a pair.
208          */
209         private boolean determineIfPair(){
210             for (int i=0; i<MAX_CARDS_IN_HAND - 1; i++){
211                 for (int j=(i + 1); j<MAX_CARDS_IN_HAND
    ; j++){
212                     if (this.ranksOfHand().get(i).equals
    (this.ranksOfHand().get(j))){
213                         return true;
214                     }
215                 }
216             }
217             return false;
218         }
219 
220         /**
221          * Identifies a hand as a twoPair.
222          * @return true or false, depending on if hand
    is a twoPair.
223          */
224         private boolean determineIfTwoPair() {
225             int matchingCard1 = 0;
226             int matchingCard2 = 0;
227             boolean x = false;
228             ArrayList<Integer> hand = new ArrayList<>();
229             hand = this.ranksOfHand();
230 
231             for (int i = 0; i < MAX_CARDS_IN_HAND - 1; i
    ++) {
232                 for (int j=(i + 1); j<MAX_CARDS_IN_HAND
    ; j++) {
233                     if (this.ranksOfHand().get(i).equals
    (this.ranksOfHand().get(j))){
234                         matchingCard1 = hand.get(i);
235                         matchingCard2 = hand.get(j);
```

```
236                     x = true;
237                 }
238             }
239         }
240         if (x){
241             hand.remove((Object) matchingCard1);
242             hand.remove((Object) matchingCard2);
243         }
244         else {
245             return false;
246         }
247 
248         for (int i = 0; (i < hand.size() - 1); i
    ++){ //2
249             for (int j = (i + 1); (j < hand.size
    ()); j++){ //3
250                 if (hand.get(i) == hand.get(j)){
251                     hand.add(matchingCard1);
252                     hand.add(matchingCard2);
253                     return true;
254                 }
255             }
256         }
257         hand.add(matchingCard1);
258         hand.add(matchingCard2);
259         return false;
260     }
261 
262     /**
263      * Identifies a highCard. Always goes off if
    none of the other card types
264      * are identifed.
265      * @return always returns true when ran.
266      */
267     private boolean determineIfHighCard(){
268         return true;
269     }
270 
271     /**
272      * Only goes off when the hand is a flush.
273      * @return returns the highest rank of a flush.
```

```
274      */
275     private ArrayList rankFlush(){
276 
277         ArrayList<Card> flush = new ArrayList<>();
278         flush = this.cardList;
279         ArrayList<Integer> ranksOfFlush = new
    ArrayList<>();
280 
281         for (int rank = HIGHEST_RANK; rank >= 0;rank
    --){ // backwards through RANKS
282             for(int card = 0; card<flush.size();
    card++){ // through each card in the flush
283                 Card cardToRank = flush.get(card);
284                 int cardRank = cardToRank.getRank();
285                 if (cardRank == rank){
286                     ranksOfFlush.add(cardRank);
287                 }
288             }
289             if (ranksOfFlush.size() ==
    MAX_CARDS_IN_HAND){ // for efficency purposes
290                 return ranksOfFlush;
291             }
292         }
293 
294         return null; // here for syntax purposes,
    will not go off.
295 
296     }
297 
298     /**
299      * Only goes off when the hand is a pair.
300      * @return Returns an arrayList of ints
    representing the hand's ranks.
301      */
302     private ArrayList<Integer> rankPair() {
303 
304         ArrayList<Integer> hand = new ArrayList<>();
305         hand = this.ranksOfHand(); // our
    unmoddified hand
306 
307         ArrayList<Integer> rankList = new ArrayList
```

```
307 <>(); // our new list of (int) ranks that we want to
     return
308         int pairRank1 = 0;
309         int pairRank2 = 0;
310 
311         for (int i = 0; i < MAX_CARDS_IN_HAND - 1; i
    ++) {
312             for (int j = (i + 1); j <
    MAX_CARDS_IN_HAND; j++) {
313                 if (this.ranksOfHand().get(i).equals
    (this.ranksOfHand().get(j))) { // pair detection
314                     pairRank1 = hand.get(i);
315                     pairRank2 = hand.get(j);
316                 }
317             }
318         }
319 
320         // append our pair's rank to rankList, &
    remove pair from the hand
321         rankList.add(pairRank1);
322         hand.remove((Object) pairRank1);
323         hand.remove((Object) pairRank2);
324 
325         for (int rank = HIGHEST_RANK; rank >= 0;rank
    --){
326             for (int card = 0; card<hand.size();
    card++){
327                 int cardToRank = hand.get(card);
328                 if (cardToRank == rank){
329                     rankList.add(cardToRank);
330                 }
331             }
332             if (rankList.size() == (
    MAX_CARDS_IN_HAND - 1)){ // for efficeincy purposes
333                 return rankList;
334             }
335         }
336 
337         return null; // will not go off, here for
    syntax purposes
338 
```

```
339     }
340 
341     /**
342      * Only goes off when the hand is a pair.
343      * @return Returns an arrayList of ints
    representing the hand's ranks.
344      */
345     private ArrayList<Integer> rankTwoPair(){
346 
347         ArrayList<Integer> hand = new ArrayList<>();
348         hand = this.ranksOfHand(); // our
    unmoddified hand
349 
350         ArrayList<Integer> rankList = new ArrayList
    <>(); // our new list of (int) ranks that we want to
     return
351         int pairRank1 = 0;
352         int pairRank2 = 0;
353 
354         int pairRank3 = 0;
355         int pairRank4 = 0;
356 
357         for (int i = 0; i < MAX_CARDS_IN_HAND - 1; i
    ++) {
358             for (int j = (i + 1); j <
    MAX_CARDS_IN_HAND; j++) {
359                 if (this.ranksOfHand().get(i).equals
    (this.ranksOfHand().get(j))) { //pair detection
360                     pairRank1 = hand.get(i);
361                     pairRank2 = hand.get(j);
362                 }
363             }
364         }
365 
366         rankList.add(pairRank1);
367         hand.remove((Object) pairRank1);
368         hand.remove((Object) pairRank2);
369 
370         for (int i = 0; i < (hand.size() - 1); i++){
371             for (int j = (i + 1); j < hand.size(); j
    ++){
```

```
372                 if (hand.get(i) == hand.get(j)){ // pair detection
373                     pairRank3 = hand.get(i);
374                     pairRank4 = hand.get(j);
375                 }
376             }
377         }
378 
379         if (rankList.get(0) > pairRank3){ // orders the two pair ranks
380             rankList.add(pairRank3);
381         }
382         else {
383             rankList.add(0, pairRank3);
384         }
385 
386         hand.remove((Object) pairRank3);
387         hand.remove((Object) pairRank4);
388         rankList.add(hand.get(0));
389 
390         return rankList;
391     }
392 
393     /**
394      * Only goes off when the hand is a 'HighCard'.
395      * @return Returns the highest Rank of the hand.
396      */
397     private ArrayList<Integer> rankHighCard(){
398 
399         ArrayList<Card> hand = new ArrayList<>();
400         hand = this.cardList;
401         ArrayList<Integer> ranksOfHighCard = new ArrayList<>();
402 
403         for (int rank = HIGHEST_RANK; rank >= 0; rank--){
404             for (int card = 0; card<hand.size(); card++){
405                 Card cardToRank = hand.get(card);
406                 int cardRank = cardToRank.getRank();
407                 if (cardRank == rank){
```

```
408                         ranksOfHighCard.add(cardRank);
409                     }
410                 }
411                 if (ranksOfHighCard.size() ==
    MAX_CARDS_IN_HAND){ // for efficency purposes
412                     return ranksOfHighCard;
413                 }
414             }
415             return null; // here for syntax purposes,
    will not go off.
416         }
417 
418 }
419 
```

```java
package proj4;

public class CardTester {
    public static void main(String[] args) {
        Testing.startTests();
        testToStringINT();
        testToStringSTR();
        testGetRank1();
        testGetRank2();
        testGetSuit1();
        testGetSuit2();
        Testing.finishTests();
    }

    public static void testToStringINT(){
        Card kingDiamond = new Card(12,3);

        Testing.assertEquals("Tests toString for a Queen of diamonds (INT)"
        , "Queen of Diamonds"
        , kingDiamond.toString());
    }

    public static void testToStringSTR(){
        Card threeDiamond = new Card("3","Clubs");

        Testing.assertEquals("Tests toString for a 3 of Clubs (STR)"
                , "3 of Clubs"
                , threeDiamond.toString());
    }

    public static void testGetRank1(){
        Card threeDiamond = new Card("three","diamonds");

        Testing.assertEquals("Tests toString for the rank of a 3 of diamonds(STR)"
                , 3
                , threeDiamond.getRank());
    }
```

```
38
39      public static void testGetRank2(){
40          Card threeDiamond = new Card("jack","Spades"
    );
41
42          Testing.assertEquals("Tests toString for the
    rank of a Jack of Spades(STR)"
43                  , 11
44                  , threeDiamond.getRank());
45      }
46
47      public static void testGetSuit1(){
48          Card eightClubs = new Card(8,2);
49
50          Testing.assertEquals("Tests toString for the
    suit of an Eight of Clubs(STR)"
51                  , "Clubs"
52                  , eightClubs.getSuit());
53      }
54
55      public static void testGetSuit2(){
56          Card eightClubs = new Card(14,1);
57
58          Testing.assertEquals("Tests toString for the
    suit of an Ace of Hearts(STR)"
59                  , "Hearts"
60                  , eightClubs.getSuit());
61      }
62  }
63
```

```java
package proj4;

public class DeckTester {

    //Deck is constructed by INTS RANK 2-14, SUIT 0-3 for each.
    // Alternatively, you can use STRINGS "#" and " suit"
    public static void main(String[] args) {
        Testing.startTests();
        testToString();
        testToStringLast3();
        testShuffle();
        testIsEmpty();
        testGather();
        testSize();
        Testing.finishTests();
    }

    public static void testToString(){
        Deck deck1 = new Deck();
        final int CARDS_TO_SKIP = 0;
        for (int x = 0; x<CARDS_TO_SKIP; x++){
            deck1.deal();
        }
        Testing.assertEquals("Tests a printout of the entire deck"
                ,"The entire deck, Spades->Clubs->Hearts->Diamonds for ranks 2-Ace"
                , deck1.toString());
    }

    public static void testToStringLast3(){
        Deck deck1 = new Deck();
        final int CARDS_TO_SKIP = 49;
        for (int x = 0; x<CARDS_TO_SKIP; x++){
            deck1.deal();
        }
        Testing.assertEquals("Tests a printout of the last 3 cards of the deck"
                ,"[Ace of Hearts, Ace of Clubs, Ace
```

```
36 of Diamonds]"
37                 , deck1.toString());
38     }
39 
40     public static void testShuffle(){
41         Deck deck1 = new Deck();
42         final int CARDS_TO_SKIP = 0;
43         for (int x = 0; x<CARDS_TO_SKIP; x++){
44             deck1.deal();
45         }
46         deck1.shuffle();
47         Testing.assertEquals("Tests a printout of the
    entire deck, shuffled"
48                 ,"52 cards in random order"
49                 , deck1.toString());
50     }
51 
52     public static void testIsEmpty(){
53         Deck deck1 = new Deck();
54         final int CARDS_TO_SKIP = 52;
55         for (int x = 0; x<CARDS_TO_SKIP; x++){
56             deck1.deal();
57         }
58         Testing.assertEquals("Tests if the deck is
   empty"
59                 ,true
60                 , deck1.isEmpty());
61     }
62 
63     public static void testGather(){
64         Deck deck1 = new Deck();
65         final int CARDS_TO_SKIP = 52;
66         for (int x = 0; x<CARDS_TO_SKIP; x++){
67             deck1.deal();
68         }
69 
70         deck1.gather();
71 
72         Testing.assertEquals("Tests if the deck is
   refilled"
73                 ,false
```

```
74                  , deck1.isEmpty());
75      }
76
77      public static void testSize(){
78          Deck deck1 = new Deck();
79          final int CARDS_TO_SKIP = 30;
80          for (int x = 0; x<CARDS_TO_SKIP; x++){
81              deck1.deal();
82          }
83          Testing.assertEquals("Tests the deck's
    remaining cards"
84                  ,22
85                  , deck1.size());
86      }
87
88  }
89
```

```java
package proj4;

import java.util.ArrayList;

public class StudPokerHand {

    private final int MAX_CARDS_IN_STUD = 2;
    private final int MAX_CARDS_IN_HAND = 5;

    private ArrayList<Card> cardList;
    private CommunityCardSet cCards;

    /**
     * Constructor.
     * @param cc The set of Community Cards currently in play.
     * @param cardList the StudPokerHand we are constructing.
     */
    public StudPokerHand(CommunityCardSet cc, ArrayList<Card> cardList){
        this.cardList = cardList;
        this.cCards = cc;
    }

    /**
     * Adds a card to the StudHand.
     * If the hand is full, does nothing.
     * @param CardObject The card that will be added.
     */
    public void addCard(Card CardObject){
        if (cardList.size() <= MAX_CARDS_IN_STUD){
            cardList.add(CardObject);
        }
        else {
            return;
        }
    }

    /**
     * Returns a card from the Stud.
```

```
39      * @param index The index of the card we are
   getting.
40      * @return returns the card at the specified
   index.
41      * If the index is invalid, returns nothing at
   all.
42      */
43     public Card get_ith_card(int index){
44 
45         if ((index >= 0) || (index <
   MAX_CARDS_IN_STUD)){
46             return cardList.get(index);
47         }
48         else {
49             return null;
50         }
51     }
52 
53     /**
54      * Returns all the cards in the Stud.
55      * @return Returns an easily readable string of
   all the cards in the Stud.
56      */
57     public String toString(){
58 
59         ArrayList<Card> studList = new ArrayList<>();
60         for (int card=0; card<cardList.size(); card
   ++){
61             studList.add( cardList.get(card) );
62         }
63         String studCards = studList.toString();
64         return studCards;
65     }
66 
67     /**
68      * Determines how this hand compares to another
   hand, using the
69      * community card set to determine the best 5-
   card hand it can
70      * make. Returns positive, negative, or zero
   depending on the comparison.
```

```
71       *
72       * @param other The hand to compare this hand to
73       * @return a negative number if this is worth
   LESS than other, zero
74       * if they are worth the SAME, and a positive
   number if this is worth
75       * MORE than other
76       */
77      public int compareTo(StudPokerHand other){
78
79          PokerHand hand1 = this.getBestFiveCardHand
   ();
80          PokerHand hand2 = other.getBestFiveCardHand
   ();
81
82          return hand1.compareTo(hand2);
83
84      }
85
86      /**
87       * Determines every possible hand that can be
   created from a stud and the
88       * available community cards, and returns a list
    of them.
89       * @return An ArrayList of all Pokerhands made
   from the 7 cards available.
90       */
91      private ArrayList<PokerHand> getAllFiveCardHands
   (){
92          ArrayList<PokerHand> handList = new
   ArrayList<>();
93
94          for (int i = 0; i < 6; i++){ // elimination
   loop
95              for (int j = (i + 1); j < 6; j++){
96
97                  ArrayList<Card> sevenList = new
   ArrayList<>(); //our seven cards
98
99                  for (int index = 0; index <
   MAX_CARDS_IN_HAND; index++){
```

```
100                     sevenList.add(cCards.
    get_ith_card(index));
101                 }
102                 for (int index = 0; index <
    MAX_CARDS_IN_STUD; index++){
103                     sevenList.add(cardList.get(index
    ));
104                 }
105                 sevenList.remove(j);
106                 sevenList.remove(i);
107                 PokerHand fiveList = new PokerHand(
    sevenList);
108                 handList.add(fiveList);
109             }
110         }
111         return handList; // contains 21 combinations
112     }
113 
114     /**
115      * Determines the best possible hand that can be
     made.
116      * @return The best possible hand that can be
    created from a stud and
117      * the available community cards.
118      */
119     private PokerHand getBestFiveCardHand()
120     {
121         ArrayList<PokerHand> hands =
    getAllFiveCardHands();
122         PokerHand bestSoFar = hands.get(0);
123         for (int i = 1; i < hands.size(); i++) {
124             if (hands.get(i).compareTo(bestSoFar) >
    0) {
125                 bestSoFar = hands.get(i);
126             }
127         }
128         return bestSoFar;
129     }
130 
131 }
132 
```

```java
package proj4;

/*
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.
 */

import java.util.ArrayList;

public class PokerHandTester {

    public static void main(String[] args) {
        //calls go here
        //CHECK VERBOSITY BOOLEAN IN TESTING//

        Testing.startTests();
        testCompareToHC1();
        testCompareTo2Flushes1();
        testCompareTo2Pairs1();
        testCompareTo2TwoPairs1();
        testCompareTo2Pairs2();
        testCompareTo2twoPairs2();
        testGetIthCard();
        testToString();
        testAddCard();
        Testing.finishTests();

    }
    //methods go here

    public static void testCompareToHC1(){
        Card c1 = new Card("11","Hearts");
        Card c2 = new Card("7","Hearts");
        Card c3 = new Card("14","Hearts");
        Card c4 = new Card("9","Diamonds");
        Card c5 = new Card("3","Hearts");
        ArrayList<Card> array1 = new ArrayList<>();
        array1.add(c1);
        array1.add(c2);
```

```
40          array1.add(c3);
41          array1.add(c4);
42          array1.add(c5);
43          PokerHand hand1 = new PokerHand(array1);
44
45          Card d1 = new Card(8,0);
46          Card d2 = new Card(7,3);
47          Card d3 = new Card(13,3);
48          Card d4 = new Card(12,3);
49          Card d5 = new Card(9,3);
50          ArrayList<Card> array2 = new ArrayList<>();
51          array2.add(d1);
52          array2.add(d2);
53          array2.add(d3);
54          array2.add(d4);
55          array2.add(d5);
56          PokerHand hand2 = new PokerHand(array2);
57
58          Testing.assertEquals("Tests for 2 highCards"
59                  , 1
60                  , hand1.compareTo(hand2));
61      }
62
63      public static void testCompareTo2Flushes1(){
64          Card c1 = new Card("14","Hearts");
65          Card c2 = new Card("12","Hearts");
66          Card c3 = new Card("12","Hearts");
67          Card c4 = new Card("11","Hearts");
68          Card c5 = new Card("7","Hearts");
69          ArrayList<Card> array1 = new ArrayList<>();
70          array1.add(c1);
71          array1.add(c2);
72          array1.add(c3);
73          array1.add(c4);
74          array1.add(c5);
75          PokerHand hand1 = new PokerHand(array1);
76
77          Card d1 = new Card(14,3);
78          Card d2 = new Card(12,3);
79          Card d3 = new Card(9,3);
80          Card d4 = new Card(9,3);
```

```
 81          Card d5 = new Card(13,3);
 82          ArrayList<Card> array2 = new ArrayList<>();
 83          array2.add(d1);
 84          array2.add(d2);
 85          array2.add(d3);
 86          array2.add(d4);
 87          array2.add(d5);
 88          PokerHand hand2 = new PokerHand(array2);
 89
 90          Testing.assertEquals("Tests 2 flushes for
    the 2nd highest card"
 91                  , -1
 92                  , hand1.compareTo(hand2));
 93      }
 94
 95
 96      public static void testCompareTo2Pairs1(){
 97          Card c1 = new Card(12,2);
 98          Card c2 = new Card(12,3);
 99          Card c3 = new Card(6,2);
100          Card c4 = new Card(5,2);
101          Card c5 = new Card(3,2);
102          ArrayList<Card> array1 = new ArrayList<>();
103          array1.add(c1);
104          array1.add(c2);
105          array1.add(c3);
106          array1.add(c4);
107          array1.add(c5);
108          PokerHand hand1 = new PokerHand(array1);
109
110          Card d1 = new Card("11","Hearts");
111          Card d2 = new Card("11","Diamonds");
112          Card d3 = new Card("9","Diamonds");
113          Card d4 = new Card("8","Diamonds");
114          Card d5 = new Card("7","Diamonds");
115          ArrayList<Card> array2 = new ArrayList<>();
116          array2.add(d1);
117          array2.add(d2);
118          array2.add(d3);
119          array2.add(d4);
120          array2.add(d5);
```

```
121         PokerHand hand2 = new PokerHand(array2);
122 
123         Testing.assertEquals("Tests CompareTo for a
    pair of queens vs a pair of jacks"
124                 , 1
125                 , hand1.compareTo(hand2));
126     }
127 
128     public static void testCompareTo2TwoPairs1(){
129         Card c1 = new Card("10","Hearts");
130         Card c2 = new Card("10","Diamonds");
131         Card c3 = new Card("8","Hearts");
132         Card c4 = new Card("8","Hearts");
133         Card c5 = new Card("3","Hearts");
134         ArrayList<Card> array1 = new ArrayList<>();
135         array1.add(c1);
136         array1.add(c2);
137         array1.add(c3);
138         array1.add(c4);
139         array1.add(c5);
140         PokerHand hand1 = new PokerHand(array1);
141 
142         Card d1 = new Card("10","Hearts");
143         Card d2 = new Card("10","Diamonds");
144         Card d3 = new Card("6","Diamonds");
145         Card d4 = new Card("6","Diamonds");
146         Card d5 = new Card("7","Diamonds");
147         ArrayList<Card> array2 = new ArrayList<>();
148         array2.add(d1);
149         array2.add(d2);
150         array2.add(d3);
151         array2.add(d4);
152         array2.add(d5);
153         PokerHand hand2 = new PokerHand(array2);
154 
155         Testing.assertEquals("Tests CompareTo for a
    twopair of 10s vs a twopair of 8s"
156                 , 1
157                 , hand1.compareTo(hand2));
158     }
159 
```

```
160     public static void testCompareTo2Pairs2(){
161         Card c1 = new Card(10,2);
162         Card c2 = new Card(10,3);
163         Card c3 = new Card(13,2);
164         Card c4 = new Card(8,2);
165         Card c5 = new Card(3,2);
166         ArrayList<Card> array1 = new ArrayList<>();
167         array1.add(c1);
168         array1.add(c2);
169         array1.add(c3);
170         array1.add(c4);
171         array1.add(c5);
172         PokerHand hand1 = new PokerHand(array1);
173 
174         Card d1 = new Card(10,2);
175         Card d2 = new Card(10,3);
176         Card d3 = new Card(11,3);
177         Card d4 = new Card(6,3);
178         Card d5 = new Card(7,3);
179         ArrayList<Card> array2 = new ArrayList<>();
180         array2.add(d1);
181         array2.add(d2);
182         array2.add(d3);
183         array2.add(d4);
184         array2.add(d5);
185         PokerHand hand2 = new PokerHand(array2);
186 
187         Testing.assertEquals("Tests for 2 pairs that
    break tie with their first chaser"
188                 , 1
189                 , hand1.compareTo(hand2));
190     }
191 
192     public static void testCompareTo2twoPairs2(){
193         Card c1 = new Card("14","Hearts");
194         Card c2 = new Card("14","Diamonds");
195         Card c3 = new Card("7","Hearts");
196         Card c4 = new Card("7","Hearts");
197         Card c5 = new Card("3","Hearts");
198         ArrayList<Card> array1 = new ArrayList<>();
199         array1.add(c1);
```

```
200         array1.add(c2);
201         array1.add(c3);
202         array1.add(c4);
203         array1.add(c5);
204         PokerHand hand1 = new PokerHand(array1);
205 
206         Card d1 = new Card("14","Spades");
207         Card d2 = new Card("14","Clubs");
208         Card d3 = new Card("7","Diamonds");
209         Card d4 = new Card("7","Diamonds");
210         Card d5 = new Card("7","Diamonds");
211         ArrayList<Card> array2 = new ArrayList<>();
212         array2.add(d1);
213         array2.add(d2);
214         array2.add(d3);
215         array2.add(d4);
216         array2.add(d5);
217         PokerHand hand2 = new PokerHand(array2);
218 
219         Testing.assertEquals("Tests for 2 twoPairs
    that break tie with their last chaser"
220                 , -1
221                 , hand1.compareTo(hand2));
222     }
223 
224     public static void testGetIthCard(){
225         Card c1 = new Card("10","Hearts");
226         Card c2 = new Card("10","Diamonds");
227         Card c3 = new Card("8","Hearts");
228         Card c4 = new Card("8","Hearts");
229         Card c5 = new Card("3","Hearts");
230         ArrayList<Card> array1 = new ArrayList<>();
231         array1.add(c1);
232         array1.add(c2);
233         array1.add(c3);
234         array1.add(c4);
235         array1.add(c5);
236         PokerHand hand1 = new PokerHand(array1);
237 
238         Testing.assertEquals("Tests getIthCard for
    the 2nd indexed card of the pokerhand"
```

```
239                 , "8 of Hearts"
240                 , hand1.get_ith_card(2));
241     }
242
243     public static void testToString(){
244         Card c1 = new Card("10","Hearts");
245         Card c2 = new Card("10","Diamonds");
246         Card c3 = new Card("8","Hearts");
247         Card c4 = new Card("8","Hearts");
248         Card c5 = new Card("3","Hearts");
249         ArrayList<Card> array1 = new ArrayList<>();
250         array1.add(c1);
251         array1.add(c2);
252         array1.add(c3);
253         array1.add(c4);
254         array1.add(c5);
255         PokerHand hand1 = new PokerHand(array1);
256
257         Testing.assertEquals("Tests toString for the
     pokerhand"
258                 , "[10 of Hearts, 10 of Diamonds, 8
    of Hearts, 8 of Hearts, 3 of Hearts]"
259                 , hand1.toString());
260     }
261
262     public static void testAddCard(){
263         Card d1 = new Card(14,3);
264         Card d2 = new Card(12,3);
265         Card d3 = new Card(9,3);
266         Card d4 = new Card(9,3);
267         Card d5 = new Card(13,3);
268         ArrayList<Card> array2 = new ArrayList<>();
269         PokerHand hand2 = new PokerHand(array2);
270         hand2.addCard(d1);
271         hand2.addCard(d2);
272         hand2.addCard(d3);
273         hand2.addCard(d4);
274         hand2.addCard(d5);
275
276         Testing.assertEquals("Tests 2 flushes for
    the 2nd highest card"
```

```
277                     , "[Ace of Diamonds, Queen of
    Diamonds, 9 of Diamonds, 9 of Diamonds, King of
    Diamonds]"
278                     , hand2.toString());
279     }
280
281 }
282
283
```

```java
package proj4;

import java.util.ArrayList;

public class CommunityCardSet {

    private final int MAX_CARDS_IN_HAND = 5;
    private ArrayList<Card> cCards;

    /**
     * Constructor.
     * @param cCards An arrayList of community cards
     * any hand can make use of.
     */
    public CommunityCardSet(ArrayList<Card> cCards){
        this.cCards = cCards;
    }

    /**
     * Adds a card to the cCard Set.
     * If the set is full, does nothing.
     * @param CardObject The card that will be added.
     */
    public void addCard(Card CardObject){

        if (cCards.size() <= MAX_CARDS_IN_HAND){
            cCards.add(CardObject);
        }
        else {
            return;
        }
    }

    /**
     * Returns a card from the set.
     * @param index The index of the card we are
getting.
     * @return returns the card at the specified
index.
     * If the index is invalid, returns nothing at
all.
```

```
39      */
40     public Card get_ith_card(int index){
41 
42         if ((index >= 0) || (index < 
   MAX_CARDS_IN_HAND)){
43             return cCards.get(index);
44         }
45         else {
46             return null;
47         }
48     }
49 
50     /**
51      * Returns all the cards in the communityCard set
   .
52      * @return Returns an easily readable string of
   all the cards in the set.
53      */
54     public String toString(){
55 
56         ArrayList<Card> handList = new ArrayList<>();
57         for (int card=0; card<cCards.size(); card++){
58             handList.add( cCards.get(card) );
59         }
60         return handList.toString();
61     }
62 
63 }
64 
```

```java
package proj4;

import java.util.ArrayList;

public class StudPokerHandTester {

    public static void main(String[] args) {
        Testing.startTests();
        testToString();
        testAddCard();
        testGetIthCard();
        Testing.finishTests();
    }

    public static void testToString(){
        Card c1 = new Card("11","Hearts");
        Card c2 = new Card("7","Hearts");
        Card c3 = new Card("14","Hearts");
        Card c4 = new Card("9","Diamonds");
        Card c5 = new Card("3","Hearts");

        Card d1 = new Card(14,3);
        Card d2 = new Card(12,3);

        ArrayList<Card> array1 = new ArrayList<>();
        CommunityCardSet cCards = new CommunityCardSet(array1);
        ArrayList<Card> array2 = new ArrayList<>();
        array2.add(d1);
        array2.add(d2);

        StudPokerHand hand2 = new StudPokerHand(cCards, array2);

        Testing.assertEquals("Tests toString of the stud"
                , "[Ace of Diamonds, Queen of Diamonds]"
                , hand2.toString());
    }

```

```
38      public static void testAddCard(){
39          Card c1 = new Card("11","Hearts");
40          Card c2 = new Card("7","Hearts");
41          Card c3 = new Card("14","Hearts");
42          Card c4 = new Card("9","Diamonds");
43          Card c5 = new Card("3","Hearts");
44
45          Card d1 = new Card(4,0);
46          Card d2 = new Card(11,2);
47
48          ArrayList<Card> array1 = new ArrayList<>();
49          CommunityCardSet cCards = new
    CommunityCardSet(array1);
50          ArrayList<Card> array2 = new ArrayList<>();
51
52          StudPokerHand hand2 = new StudPokerHand(
    cCards, array2);
53          hand2.addCard(d1);
54          hand2.addCard(d2);
55
56          Testing.assertEquals("Tests addCard of the
    stud"
57                  , "[4 of Spades, Jack of Clubs]"
58                  , hand2.toString());
59      }
60
61      public static void testGetIthCard(){
62          Card c1 = new Card("11","Hearts");
63          Card c2 = new Card("7","Hearts");
64          Card c3 = new Card("14","Hearts");
65          Card c4 = new Card("9","Diamonds");
66          Card c5 = new Card("3","Hearts");
67
68          Card d1 = new Card(14,3);
69          Card d2 = new Card(12,3);
70
71          ArrayList<Card> array1 = new ArrayList<>();
72          CommunityCardSet cCards = new
    CommunityCardSet(array1);
73          ArrayList<Card> array2 = new ArrayList<>();
74          array2.add(d1);
```

```
75          array2.add(d2);
76
77          StudPokerHand hand2 = new StudPokerHand(
    cCards, array2);
78
79          Testing.assertEquals("Tests getIthCard of
    the stud"
80                  , "Queen of Diamonds"
81                  , hand2.get_ith_card(1));
82      }
83
84  }
85
```

```java
package proj4;

import java.util.ArrayList;

public class CommunityCardSetTester {

    public static void main(String[] args) {

        Testing.startTests();
        testToString();
        testAddCard();
        testGetIthCard();
        Testing.finishTests();
    }

    public static void testToString(){
        Card c1 = new Card("11","Hearts");
        Card c2 = new Card("7","Hearts");
        Card c3 = new Card("14","Hearts");
        Card c4 = new Card("9","Diamonds");
        Card c5 = new Card("3","Hearts");
        ArrayList<Card> array1 = new ArrayList<>();
        array1.add(c1);
        array1.add(c2);
        array1.add(c3);
        array1.add(c4);
        array1.add(c5);
        PokerHand hand1 = new PokerHand(array1);

        Testing.assertEquals("Tests for communitycardsets toString"
                , "[Jack of Hearts, 7 of Hearts, Ace of Hearts, 9 of Diamonds, 3 of Hearts]"
                , hand1.toString());
    }

    public static void testAddCard(){
        Card d1 = new Card(2,0);
        Card d2 = new Card(12,3);
        Card d3 = new Card(10,2);
        Card d4 = new Card(9,1);
```

```
40          Card d5 = new Card(13,3);
41          ArrayList<Card> array2 = new ArrayList<>();
42          PokerHand hand2 = new PokerHand(array2);
43          hand2.addCard(d1);
44          hand2.addCard(d2);
45          hand2.addCard(d3);
46          hand2.addCard(d4);
47          hand2.addCard(d5);
48
49          Testing.assertEquals("Tests 2 flushes for the
    2nd highest card"
50                  , "[2 of Spades, Queen of Diamonds,
   10 of Clubs, 9 of Hearts, King of Diamonds]"
51                  , hand2.toString());
52      }
53
54      public static void testGetIthCard(){
55          Card c1 = new Card("10","Hearts");
56          Card c2 = new Card("10","Diamonds");
57          Card c3 = new Card("8","Hearts");
58          Card c4 = new Card("8","Hearts");
59          Card c5 = new Card("3","Hearts");
60          ArrayList<Card> array1 = new ArrayList<>();
61          array1.add(c1);
62          array1.add(c2);
63          array1.add(c3);
64          array1.add(c4);
65          array1.add(c5);
66          PokerHand hand1 = new PokerHand(array1);
67
68          Testing.assertEquals("Tests getIthCard for
   the 2nd indexed card of the pokerhand"
69                  , "8 of Hearts"
70                  , hand1.get_ith_card(2));
71      }
72  }
73
```