

```
1 package proj4;
2
3 public class Card {
4
5     private int rank;
6     private int suit;
7
8     private final String[] strNumRanks = {"  
fillerIndex0", "fillerIndex1", "2", "3", "4", "5",  
         "6", "7", "8", "9", "10", "11", "12", "13"  
", "14"};
9     private final String[] strStringRanks = {"  
fillerIndex0", "fillerIndex1", "Two", "Three", "Four"  
, "Five",
10         "Six", "Seven", "Eight", "Nine", "Ten", "  
Eleven", "Twelve", "Thirteen", "Fourteen"};
11     private final String[] strUpperRanks = {"  
fillerIndex0", "fillerIndex1", "Two", "Three", "Four"  
, "Five",
12         "Six", "Seven", "Eight", "Nine", "Ten", "  
Jack", "Queen", "King", "Ace"};
13
14     private final String[] strStringSuits = {"Spades"  
, "Hearts", "Clubs", "Diamonds"};
15
16     /**
17      * constructor
18      * @param rank String: whole cards (2-10) can
19      * either be spelled
20      * out like "two" or numeric like "2". Case
21      * insensitive.
22      * @param suit String: "Spades", "Hearts", "Clubs"
23      * , or "Diamonds"
24      */
25     public Card(String rank, String suit){
26
27         this.rank = rankString2rankInt(rank);
28         this.suit = suitString2suitInt(suit);
29     }
30 }
```

```
30     /**
31      * constructor
32      * @param rank integer between 2-14
33      * @param suit integer: 0=Spades, 1=Hearts, 2=
34      * Clubs, or 3=Diamonds
35     */
36     public Card(int rank, int suit){
37
38         this.rank = rank;
39         this.suit = suit;
40     }
41
42
43     /**
44      * Converts a String rank to an integer.
45      * @param rank2convert The rank we are converting
46      * to an integer.
47      * @return The int the rank represents
48     */
49     private int rankString2rankInt(String
50         rank2convert){
51
52         for (int index = 2; index < strNumRanks.
53             length; index++) {
54             if ((rank2convert.equalsIgnoreCase(
55                 strNumRanks[index])) ||
56                 (rank2convert.equalsIgnoreCase(
57                     strStringRanks[index])) ||
58                 (rank2convert.equalsIgnoreCase(
59                     strUpperRanks[index])))
60             {
61                 return index;
62             }
63         }
64         return 999; // will not go off, here for
65         syntax purposes
66     }
67
68     /**
69      * 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
97             syntax purposes
98
99     /**
100      * Uses getRank() to get an int Rank, then
101      converts it back into a String.
102      * @return A stringified version of the int Rank
103
104
105     private String rank2String(){
106         int myRank = getRank();
107         if (myRank == 11) {
108             return "Jack";
109         }
110         else if (myRank == 12) {
111             return "Queen";
112         }
113         else if (myRank == 13) {
114             return "King";
115         }
116         else if (myRank == 14) {
117             return "Ace";
118         }
119         else { // must be a whole number - just
120             convert it to string
121             return "" + myRank;
122         }
123
124     /**
125      * returns string version of Card
126      * @return string like "Jack of Clubs"
127      */
128     public String toString() {
129         return rank2String() + " of " + getSuit();
130     }
131 }
```

```
1 package proj4;
2
3 import java.util.ArrayList;
4 import java.util.concurrent.ThreadLocalRandom;
5
6 public class Deck {
7
8     private final int[] RANKS = {2, 3, 4, 5, 6, 7, 8
9 , 9, 10, 11, 12, 13, 14};
10    private final int[] SUITS = {0,1,2,3};
11    private final int MAX_CARDS_IN_DECK = 52;
12
13    private ArrayList<Card> cardDeck;
14    private int nextToDeal;
15
16    /**
17     *constructor. Makes a deck of 52 playing cards
18     */
19    public Deck(){
20
21         this.cardDeck = new ArrayList<>(
22             MAX_CARDS_IN_DECK);
23         this.nextToDeal = 0;
24
25         for (int rankIndex=0; rankIndex<RANKS.length
26 ; rankIndex++){
27             for(char suitIndex=0; suitIndex<SUITS.
28                 length; suitIndex++){
29                 cardDeck.add( new Card(RANKS[
30                     rankIndex],SUITS[suitIndex]) );
31             }
32         }
33     }
34
35     /**
36      * Shuffles the Deck. Only shuffles cards that
37      * have NOT been dealt.
38      */
39     public void shuffle(){
40         for (int cardIndex=0; cardIndex<cardDeck.size
41 (); cardIndex++){
42             int randomIndex = ThreadLocalRandom.
43                 current().nextInt(cardIndex);
44             Card tempCard = cardDeck.get(cardIndex);
45             cardDeck.set(cardIndex, cardDeck.get(randomIndex));
46             cardDeck.set(randomIndex, tempCard);
47         }
48     }
49 }
```

```

35         int randomIndex = ThreadLocalRandom.
36             current().nextInt(nextToDeal, MAX_CARDS_IN_DECK);
37             cardDeck = swap(cardDeck, cardIndex,
38             randomIndex);
39     }
40
41     /**
42      * A method to swap all the undealt cards in a
43      * @param list an ArrayList of <Card> type</Card>
44      * @param currentIndicy the indicy of the current
45      * card in our loop
46      * @param randomIndicy the indicy of a random
47      * undealt card in the deck
48      * @return returns a swapped ArrayList (
49      * specifically a swapped deck)
50      */
51     private ArrayList<Card> swap(ArrayList<Card> list
52 , int currentIndicy, int randomIndicy){
53
54         Card temp1;
55         Card temp2;
56
57         temp1 = list.get(currentIndicy);
58         temp2 = list.get(randomIndicy);
59
60         list.set(currentIndicy,temp2);
61         list.set(randomIndicy, temp1);
62
63         return list;
64     }
65
66     /**
67      * Deals a card, or returns nothing if the deck
68      * is empty.
69      * @return Returns a card, or null the deck is
70      * empty.
71      */
72     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
78      * the deck.
79      * @return true or false, depending on if the
80      * deck has cards left to deal.
81      */
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
90      * not been dealt yet.
91      * @return integer number of cards not yet dealt
92      */
92     public int size(){
93         return (MAX_CARDS_IN_DECK - nextToDeal);
94     }
95
96     /**
97      * Returns the deck to a state where no cards
98      * have been dealt
99      */
99     public void gather(){
100        nextToDeal = 0;
101    }
102
```

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

```
1 package proj4;
2
3 /*
4 I affirm that I have carried out the attached
5 academic endeavors with full academic honesty, in
6 accordance with the Union College Honor Code and the
7 course syllabus.
8 */
9
10
11 import java.util.ArrayList;
12 import java.util.Scanner;
13
14 public class Client {
15     final int MAX_CARDS_IN_HAND = 5;
16     final int MAX_CARDS_IN_COMSET = 5;
17     final int MAX_CARDS_IN_STUD = 2;
18     int points = 0;
19     String messageLoss = "Incorrect. Final points
20 : " + points;
21     String messageWin = "You win! Final points: "
22 ; // + points;
23
24     Deck newDeck = new Deck();
25     newDeck.shuffle(); // create and shuffle deck
26
27     ArrayList<Card> communityArray = new
28     ArrayList<>(); // create community cards
29     CommunityCardSet cardSet = new
30     CommunityCardSet(communityArray);
31
32     ArrayList<Card> h1 = new ArrayList<>(); // // create studs
33     StudPokerHand stud1 = new StudPokerHand(
34     cardSet, h1);
35     ArrayList<Card> h2 = new ArrayList<>();
36     StudPokerHand stud2 = new StudPokerHand(
37     cardSet, h2);
38
39     System.out.println("Community cards: " +
40     communityArray);
41
42     System.out.println("Hand 1: " + h1);
43     System.out.println("Hand 2: " + h2);
44
45     if (stud1.getScore() > stud2.getScore()) {
46         System.out.println("Hand 1 wins!");
47     } else if (stud2.getScore() > stud1.getScore()) {
48         System.out.println("Hand 2 wins!");
49     } else {
50         System.out.println("It's a tie!");
51     }
52 }
```

```

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
stud 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 +
67             points);
68             System.out.println("The answer was
69             : " + answer);
70             return;
71         }
72         h1.clear();
73         h2.clear();
74     }
75     System.out.println(messageWin + points);
76 }
77 }
78 }
79 }
```

```

1 package proj4;
2
3 /**
4  * This class contains a collection of methods that
5  * help with testing. All methods
6  * here are static so there's no need to construct a
7  * Testing object. Just call them
8  * with the class name like so:
9  * <p></p>
10 * <code>Testing.assertEquals("test description",
11 * expected, actual)</code>
12 *
13 * @author Kristina Striegnitz, Aaron Cass, Chris
14 * Fernandes
15 * @version 5/28/18
16 */
17
18 public class Testing {
19
20     private static boolean VERBOSE = true;
21     private static int numTests;
22     private static int numFails;
23
24     /**
25      * Toggles between a lot of output and little
26      * output.
27      *
28      * @param verbose
29      *           If verbose is true, then complete
30      *           information is printed,
31      *           whether the tests passes or fails.
32      *           If verbose is false, only
33      *           failures are printed.
34     */
35
36     public static void setVerbose(boolean verbose)
37     {
38         VERBOSE = verbose;
39     }
40
41     /**
42      * Each of the assertEquals methods tests whether
43      * 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,
72                                     Object expected,
73                                     Object actual)
74     {
75         String expectedString = "<<null>>";
76         String actualString = "<<null>>";
77         if (expected != null) {
78             expectedString = expected.toString();
79         }
80         if (actual != null) {
81             actualString = actual.toString();
82         }
83         printTestCaseInfo(message, expectedString,
84                           actualString);
85
86         if (expected == null) {
87             if (actual == null) {
88                 pass();
89             } else {
90                 fail(message);
91             }
92         } else if (expected.equals(actual)) {
93             pass();
94         } else {
95             fail(message);
96         }
97     /**
98      * Asserts that a given boolean must be true.
99      * The test fails if
100      *   the boolean is not true.
101      *   @param message The test message
102      *   @param actual The boolean value asserted to
103      *   be true.
104      */
105 }
```

```

104     public static void assertTrue(String message,
105         boolean actual)
106     {
107         assertEquals(message, true, actual);
108     }
109     /**
110      * Asserts that a given boolean must be false.
111      * The test fails if
112      * the boolean is not false (i.e. if it is true
113      *).
114      *
115      * @param message The test message
116      * @param actual The boolean value asserted to
117      * be false.
118      */
119     public static void assertFalse(String message,
120         boolean actual)
121     {
122         assertEquals(message, false, actual);
123     }
124     private static void printTestCaseInfo(String
125         message, String expected,
126                                         String
127         actual)
128     {
129         if (VERBOSE) {
130             System.out.println(message + ":");
131             System.out.println("expected: " +
132                 expected);
133             System.out.println("actual:    " + actual
134 );
135             }
136         }
137     private static void pass()
138     {
139         numTests++;
140
141         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
157      * printed.
158      */
159     public static void testSection(String
160                                   sectionTitle)
161     {
162         if (VERBOSE) {
163             int dashCount = sectionTitle.length();
164             System.out.println(sectionTitle);
165             for (int i = 0; i < dashCount; i++) {
166                 System.out.print("-");
167             }
168         }
169     }
170
171     /**
172      * Initializes the test suite. Should be called
173      * before running any
174      * 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.  
185      * Should be called  
186      * after all the tests have run.  
187      */  
188     public static void finishTests()  
189     {  
190         System.out.println("=====");  
191         System.out.println("Tests Complete");  
192         System.out.println("=====");  
193         int numPasses = numTests - numFails;  
194  
195         System.out.print(numPasses + "/" + numTests  
196             + " PASS ");  
197         System.out.printf("(pass rate: %.1f%s)\n",  
198             100 * ((double) numPasses) /  
199             numTests,  
200             "%");  
201  
202         System.out.print(numFails + "/" + numTests  
203             + " FAIL ");  
204         System.out.printf("(fail rate: %.1f%s)\n",  
205             100 * ((double) numFails) / numTests  
206             ,  
207             "%");  
208     }  
209 }
```

```

1 package proj4;
2
3 import java.util.ArrayList;
4
5 public class PokerHand {
6
7     private final int[] RANKS = {2, 3, 4, 5, 6, 7, 8
8 , 9, 10, 11, 12, 13, 14};
9     private final int HIGHEST_RANK = 14;
10    private final int MAX_CARDS_IN_HAND = 5;
11    private ArrayList<Card> cardList;
12
13    /**
14     * Constructor. models a hand of 5 playing cards.
15     * @param cardList A list of cards that should be
16     * in the hand
17     */
18    public PokerHand(ArrayList<Card> cardList){
19        this.cardList = cardList;
20    }
21
22    /**
23     * Adds a card to the hand.
24     * If the hand is full, does nothing.
25     * @param CardObject The card that will be added.
26     */
27    public void addCard(Card CardObject){
28
29        if (cardList.size() <= MAX_CARDS_IN_HAND){
30            cardList.add(CardObject);
31        }
32        else {
33            return;
34        }
35
36    /**
37     * Returns a card from the hand.
38     * @param index The index of the card we are
39     * getting.

```

```

39      * @return returns the card at the specified
40      * index.
41      *
42      public Card get_ith_card(int index){
43
44          if ((index >= 0) || (index <
45              MAX_CARDS_IN_HAND)){
46              return cardList.get(index);
47          }
48          else {
49              return null;
50          }
51
52      /**
53      * Returns all the cards in a hand.
54      * @return Returns an easily readable string of
55      * all the cards in a hand.
56      */
57      public String toString(){
58
59          ArrayList<Card> handList = new ArrayList<>();
60          for (int card=0; card<cardList.size(); card
61              ++){
62              handList.add( cardList.get(card) );
63          }
64
65      /**
66      * Determines how this hand compares to another
67      * hand, returns
68      * positive, negative, or zero depending on the
69      * comparison.
70      *
71      * @param other The hand to compare this hand to
72      * @return a negative number if this is worth
73      * LESS than other, zero
74      * 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.
108              determineIfFlush()) || // flushes and HighCards
109                  ((hand1.determineIfHighCard() &&
110                      hand2.determineIfHighCard()) &&
111                          (!hand1.determineIfPair
112                              () && !hand2.determineIfPair()) &&
113                                  (!hand1.determineIfTwoPair
114                                      () && !hand2.determineIfTwoPair()))){
115
116          ArrayList<Integer> h1 = hand1.rankFlush
117          ();
118          ArrayList<Integer> h2 = hand2.rankFlush
119          ();
120
121          // systematically ranking flushes and
122          // highCards is the same
123
124          for (int card = 0; card<h1.size(); card
125              ++){ // iterates through hands
126
127              if (h1.get(card) > h2.get(card)){
128                  return 1;
129              }
130
131              else if (h1.get(card) < h2.get(card
132                  )){
133
134                  return -1;
135              }
136
137          }
138
139
140          if (hand1.determineIfTwoPair() && hand2.
141              determineIfTwoPair()){ // twoPairs
142
143              ArrayList<Integer> h1 = hand1.
144                  rankTwoPair();
145
146              ArrayList<Integer> h2 = hand2.
147                  rankTwoPair();
148
149              for (int card = 0; card<h1.size(); card
150

```

```
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 ) {
141                     return -1;
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         return 0;
161     }
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
170     * cards' ranks
171     * @return Returns an ArrayList of integers in
172     * easily readable form.
173     */
174     private ArrayList<Integer> ranksOfHand(){
175
176         ArrayList<Integer> rankList = new ArrayList
177         <>();
178         Card cardIndex;
179         int card;
180
181         for (int i = 0; i<MAX_CARDS_IN_HAND; i++){
182             cardIndex = this.get_ith_card(i);
183             card = cardIndex.getRank();
184             rankList.add(card);
185         }
186         return rankList;
187     }
188
189     /**
190      * Identifies a hand as a flush.
191      * @return true or false, depending on if hand
192      * is a flush.
193      */
194     private boolean determineIfFlush(){
195
196         Card card;
197         String cardSuit;
198
199         card = this.get_ith_card(0);
200         cardSuit = card.getSuit(); // cardSuit; the
201         suit of the first card in a hand
202         for (int cardIndex=0; cardIndex<
203             MAX_CARDS_IN_HAND; cardIndex++){ //iterates through
204             a hand
205                 if (!cardSuit.equalsIgnoreCase(this.
206                 get_ith_card(cardIndex).getSuit())){
207                     return false;
208                 }
209             }

```

```
202         return true;
203     }
204
205     /**
206      * Identifies a hand as a pair.
207      * @return true or false, depending on if hand
208      * is a pair.
209     */
210    private boolean determineIfPair(){
211        for (int i=0; i<MAX_CARDS_IN_HAND - 1; i++){
212            for (int j=(i + 1); j<MAX_CARDS_IN_HAND
213 ; j++){
214                if (this.ranksOfHand().get(i).equals
215 (this.ranksOfHand().get(j))){
216                    return true;
217                }
218            }
219        }
220        /**
221         * Identifies a hand as a twoPair.
222         * @return true or false, depending on if hand
223         * is a twoPair.
224         */
225        private boolean determineIfTwoPair() {
226            int matchingCard1 = 0;
227            int matchingCard2 = 0;
228            boolean x = false;
229            ArrayList<Integer> hand = new ArrayList<>();
230            hand = this.ranksOfHand();
231
232            for (int i = 0; i < MAX_CARDS_IN_HAND - 1; i
233 ++){
234                for (int j=(i + 1); j<MAX_CARDS_IN_HAND
235 ; j++) {
236                    if (this.ranksOfHand().get(i).equals
237 (this.ranksOfHand().get(j))){
238                        matchingCard1 = hand.get(i);
239                        matchingCard2 = hand.get(j);
240                    }
241                }
242            }
243            if (matchingCard1 == matchingCard2) {
244                x = true;
245            }
246        }
247        return x;
248    }
```

```

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
249             ++){ //2
250             for (int j = (i + 1); (j < hand.size
251             ()); j++){ //3
252                 if (hand.get(i) == hand.get(j)){
253                     hand.add(matchingCard1);
254                     hand.add(matchingCard2);
255                     return true;
256                 }
257             }
258             hand.add(matchingCard1);
259             hand.add(matchingCard2);
260             return false;
261         }
262     /**
263      * Identifies a highCard. Always goes off if
264      * none of the other card types
265      * are identified.
266      * @return always returns true when ran.
267     */
268     private boolean determineIfHighCard(){
269         return true;
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
280             ArrayList<>();
281
282         for (int rank = HIGHEST_RANK; rank >= 0;rank
283             --){ // backwards through RANKS
284             for(int card = 0; card<flush.size();
285                 card++){ // through each card in the flush
286                 Card cardToRank = flush.get(card);
287                 int cardRank = cardToRank.getRank();
288                 if (cardRank == rank){
289                     ranksOfFlush.add(cardRank);
290                 }
291             }
292         }
293
294         return ranksOfFlush.size() ==
295             MAX_CARDS_IN_HAND){ // for efficiency purposes
296             return ranksOfFlush;
297         }
298     }
299
300     /**
301      * Only goes off when the hand is a pair.
302      * @return Returns an arrayList of ints
303      * representing the hand's ranks.
304      */
305     private ArrayList<Integer> rankPair() {
306
307         ArrayList<Integer> hand = new ArrayList<>();
308         hand = this.ranksOfHand(); // our
309         unmodified hand
310
311         ArrayList<Integer> rankList = new ArrayList<>();
312
313         for (int i = 0; i < hand.size(); i++)
314             for (int j = i+1; j < hand.size(); j++)
315                 if (hand.get(i) == hand.get(j))
316                     rankList.add(hand.get(i));
317
318         return rankList;
319     }
320
321     /**
322      * Checks if the hand has a pair.
323      * @return true if there is a pair, false otherwise.
324      */
325     public boolean hasPair(){
326
327         ArrayList<Integer> rankList = rankPair();
328
329         if (rankList.size() > 0)
330             return true;
331         else
332             return false;
333     }
334
335     /**
336      * Checks if the hand has a three of a kind.
337      * @return true if there is a three of a kind, false otherwise.
338      */
339     public boolean hasThreeOfAKind(){
340
341         ArrayList<Integer> rankList = rankPair();
342
343         if (rankList.size() > 0)
344             return true;
345         else
346             return false;
347     }
348
349     /**
350      * Checks if the hand has a four of a kind.
351      * @return true if there is a four of a kind, false otherwise.
352      */
353     public boolean hasFourOfAKind(){
354
355         ArrayList<Integer> rankList = rankPair();
356
357         if (rankList.size() > 0)
358             return true;
359         else
360             return false;
361     }
362
363     /**
364      * Checks if the hand has a full house.
365      * @return true if there is a full house, false otherwise.
366      */
367     public boolean hasFullHouse(){
368
369         ArrayList<Integer> rankList = rankPair();
370
371         if (rankList.size() > 0)
372             return true;
373         else
374             return false;
375     }
376
377     /**
378      * Checks if the hand has a straight.
379      * @return true if there is a straight, false otherwise.
380      */
381     public boolean hasStraight(){
382
383         ArrayList<Integer> rankList = rankPair();
384
385         if (rankList.size() > 0)
386             return true;
387         else
388             return false;
389     }
390
391     /**
392      * Checks if the hand has a flush.
393      * @return true if there is a flush, false otherwise.
394      */
395     public boolean hasFlush(){
396
397         ArrayList<Integer> rankList = rankPair();
398
399         if (rankList.size() > 0)
400             return true;
401         else
402             return false;
403     }
404
405     /**
406      * Checks if the hand has a royal flush.
407      * @return true if there is a royal flush, false otherwise.
408      */
409     public boolean hasRoyalFlush(){
410
411         ArrayList<Integer> rankList = rankPair();
412
413         if (rankList.size() > 0)
414             return true;
415         else
416             return false;
417     }
418
419     /**
420      * Checks if the hand has a straight flush.
421      * @return true if there is a straight flush, false otherwise.
422      */
423     public boolean hasStraightFlush(){
424
425         ArrayList<Integer> rankList = rankPair();
426
427         if (rankList.size() > 0)
428             return true;
429         else
430             return false;
431     }
432
433     /**
434      * Checks if the hand has a royal straight flush.
435      * @return true if there is a royal straight flush, false otherwise.
436      */
437     public boolean hasRoyalStraightFlush(){
438
439         ArrayList<Integer> rankList = rankPair();
440
441         if (rankList.size() > 0)
442             return true;
443         else
444             return false;
445     }
446
447     /**
448      * Checks if the hand has a wild card.
449      * @return true if there is a wild card, false otherwise.
450      */
451     public boolean hasWildCard(){
452
453         ArrayList<Integer> rankList = rankPair();
454
455         if (rankList.size() > 0)
456             return true;
457         else
458             return false;
459     }
460
461     /**
462      * Checks if the hand has a wild card.
463      * @return true if there is a wild card, false otherwise.
464      */
465     public boolean hasWildCard2(){
466
467         ArrayList<Integer> rankList = rankPair();
468
469         if (rankList.size() > 0)
470             return true;
471         else
472             return false;
473     }
474
475     /**
476      * Checks if the hand has a wild card.
477      * @return true if there is a wild card, false otherwise.
478      */
479     public boolean hasWildCard3(){
480
481         ArrayList<Integer> rankList = rankPair();
482
483         if (rankList.size() > 0)
484             return true;
485         else
486             return false;
487     }
488
489     /**
490      * Checks if the hand has a wild card.
491      * @return true if there is a wild card, false otherwise.
492      */
493     public boolean hasWildCard4(){
494
495         ArrayList<Integer> rankList = rankPair();
496
497         if (rankList.size() > 0)
498             return true;
499         else
500             return false;
501     }
502
503     /**
504      * Checks if the hand has a wild card.
505      * @return true if there is a wild card, false otherwise.
506      */
507     public boolean hasWildCard5(){
508
509         ArrayList<Integer> rankList = rankPair();
510
511         if (rankList.size() > 0)
512             return true;
513         else
514             return false;
515     }
516
517     /**
518      * Checks if the hand has a wild card.
519      * @return true if there is a wild card, false otherwise.
520      */
521     public boolean hasWildCard6(){
522
523         ArrayList<Integer> rankList = rankPair();
524
525         if (rankList.size() > 0)
526             return true;
527         else
528             return false;
529     }
530
531     /**
532      * Checks if the hand has a wild card.
533      * @return true if there is a wild card, false otherwise.
534      */
535     public boolean hasWildCard7(){
536
537         ArrayList<Integer> rankList = rankPair();
538
539         if (rankList.size() > 0)
540             return true;
541         else
542             return false;
543     }
544
545     /**
546      * Checks if the hand has a wild card.
547      * @return true if there is a wild card, false otherwise.
548      */
549     public boolean hasWildCard8(){
550
551         ArrayList<Integer> rankList = rankPair();
552
553         if (rankList.size() > 0)
554             return true;
555         else
556             return false;
557     }
558
559     /**
560      * Checks if the hand has a wild card.
561      * @return true if there is a wild card, false otherwise.
562      */
563     public boolean hasWildCard9(){
564
565         ArrayList<Integer> rankList = rankPair();
566
567         if (rankList.size() > 0)
568             return true;
569         else
570             return false;
571     }
572
573     /**
574      * Checks if the hand has a wild card.
575      * @return true if there is a wild card, false otherwise.
576      */
577     public boolean hasWildCard10(){
578
579         ArrayList<Integer> rankList = rankPair();
580
581         if (rankList.size() > 0)
582             return true;
583         else
584             return false;
585     }
586
587     /**
588      * Checks if the hand has a wild card.
589      * @return true if there is a wild card, false otherwise.
590      */
591     public boolean hasWildCard11(){
592
593         ArrayList<Integer> rankList = rankPair();
594
595         if (rankList.size() > 0)
596             return true;
597         else
598             return false;
599     }
599
600     /**
601      * Checks if the hand has a wild card.
602      * @return true if there is a wild card, false otherwise.
603      */
604     public boolean hasWildCard12(){
605
606         ArrayList<Integer> rankList = rankPair();
607
608         if (rankList.size() > 0)
609             return true;
610         else
611             return false;
612     }
612
613     /**
614      * Checks if the hand has a wild card.
615      * @return true if there is a wild card, false otherwise.
616      */
617     public boolean hasWildCard13(){
618
619         ArrayList<Integer> rankList = rankPair();
620
621         if (rankList.size() > 0)
622             return true;
623         else
624             return false;
625     }
625
626     /**
627      * Checks if the hand has a wild card.
628      * @return true if there is a wild card, false otherwise.
629      */
630     public boolean hasWildCard14(){
631
632         ArrayList<Integer> rankList = rankPair();
633
634         if (rankList.size() > 0)
635             return true;
636         else
637             return false;
638     }
638
639     /**
640      * Checks if the hand has a wild card.
641      * @return true if there is a wild card, false otherwise.
642      */
643     public boolean hasWildCard15(){
644
645         ArrayList<Integer> rankList = rankPair();
646
647         if (rankList.size() > 0)
648             return true;
649         else
650             return false;
651     }
651
652     /**
653      * Checks if the hand has a wild card.
654      * @return true if there is a wild card, false otherwise.
655      */
656     public boolean hasWildCard16(){
657
658         ArrayList<Integer> rankList = rankPair();
659
660         if (rankList.size() > 0)
661             return true;
662         else
663             return false;
664     }
664
665     /**
666      * Checks if the hand has a wild card.
667      * @return true if there is a wild card, false otherwise.
668      */
669     public boolean hasWildCard17(){
670
671         ArrayList<Integer> rankList = rankPair();
672
673         if (rankList.size() > 0)
674             return true;
675         else
676             return false;
677     }
677
678     /**
679      * Checks if the hand has a wild card.
680      * @return true if there is a wild card, false otherwise.
681      */
682     public boolean hasWildCard18(){
683
684         ArrayList<Integer> rankList = rankPair();
685
686         if (rankList.size() > 0)
687             return true;
688         else
689             return false;
690     }
690
691     /**
692      * Checks if the hand has a wild card.
693      * @return true if there is a wild card, false otherwise.
694      */
695     public boolean hasWildCard19(){
696
697         ArrayList<Integer> rankList = rankPair();
698
699         if (rankList.size() > 0)
700             return true;
701         else
702             return false;
703     }
703
704     /**
705      * Checks if the hand has a wild card.
706      * @return true if there is a wild card, false otherwise.
707      */
708     public boolean hasWildCard20(){
709
710         ArrayList<Integer> rankList = rankPair();
711
712         if (rankList.size() > 0)
713             return true;
714         else
715             return false;
716     }
716
717     /**
718      * Checks if the hand has a wild card.
719      * @return true if there is a wild card, false otherwise.
720      */
721     public boolean hasWildCard21(){
722
723         ArrayList<Integer> rankList = rankPair();
724
725         if (rankList.size() > 0)
726             return true;
727         else
728             return false;
729     }
729
730     /**
731      * Checks if the hand has a wild card.
732      * @return true if there is a wild card, false otherwise.
733      */
734     public boolean hasWildCard22(){
735
736         ArrayList<Integer> rankList = rankPair();
737
738         if (rankList.size() > 0)
739             return true;
740         else
741             return false;
742     }
742
743     /**
744      * Checks if the hand has a wild card.
745      * @return true if there is a wild card, false otherwise.
746      */
747     public boolean hasWildCard23(){
748
749         ArrayList<Integer> rankList = rankPair();
750
751         if (rankList.size() > 0)
752             return true;
753         else
754             return false;
755     }
755
756     /**
757      * Checks if the hand has a wild card.
758      * @return true if there is a wild card, false otherwise.
759      */
760     public boolean hasWildCard24(){
761
762         ArrayList<Integer> rankList = rankPair();
763
764         if (rankList.size() > 0)
765             return true;
766         else
767             return false;
768     }
768
769     /**
770      * Checks if the hand has a wild card.
771      * @return true if there is a wild card, false otherwise.
772      */
773     public boolean hasWildCard25(){
774
775         ArrayList<Integer> rankList = rankPair();
776
777         if (rankList.size() > 0)
778             return true;
779         else
780             return false;
781     }
781
782     /**
783      * Checks if the hand has a wild card.
784      * @return true if there is a wild card, false otherwise.
785      */
786     public boolean hasWildCard26(){
787
788         ArrayList<Integer> rankList = rankPair();
789
790         if (rankList.size() > 0)
791             return true;
792         else
793             return false;
794     }
794
795     /**
796      * Checks if the hand has a wild card.
797      * @return true if there is a wild card, false otherwise.
798      */
799     public boolean hasWildCard27(){
800
801         ArrayList<Integer> rankList = rankPair();
802
803         if (rankList.size() > 0)
804             return true;
805         else
806             return false;
807     }
807
808     /**
809      * Checks if the hand has a wild card.
810      * @return true if there is a wild card, false otherwise.
811      */
812     public boolean hasWildCard28(){
813
814         ArrayList<Integer> rankList = rankPair();
815
816         if (rankList.size() > 0)
817             return true;
818         else
819             return false;
820     }
820
821     /**
822      * Checks if the hand has a wild card.
823      * @return true if there is a wild card, false otherwise.
824      */
825     public boolean hasWildCard29(){
826
827         ArrayList<Integer> rankList = rankPair();
828
829         if (rankList.size() > 0)
830             return true;
831         else
832             return false;
833     }
833
834     /**
835      * Checks if the hand has a wild card.
836      * @return true if there is a wild card, false otherwise.
837      */
838     public boolean hasWildCard30(){
839
840         ArrayList<Integer> rankList = rankPair();
841
842         if (rankList.size() > 0)
843             return true;
844         else
845             return false;
846     }
846
847     /**
848      * Checks if the hand has a wild card.
849      * @return true if there is a wild card, false otherwise.
850      */
851     public boolean hasWildCard31(){
852
853         ArrayList<Integer> rankList = rankPair();
854
855         if (rankList.size() > 0)
856             return true;
857         else
858             return false;
859     }
859
860     /**
861      * Checks if the hand has a wild card.
862      * @return true if there is a wild card, false otherwise.
863      */
864     public boolean hasWildCard32(){
865
866         ArrayList<Integer> rankList = rankPair();
867
868         if (rankList.size() > 0)
869             return true;
870         else
871             return false;
872     }
872
873     /**
874      * Checks if the hand has a wild card.
875      * @return true if there is a wild card, false otherwise.
876      */
877     public boolean hasWildCard33(){
878
879         ArrayList<Integer> rankList = rankPair();
880
881         if (rankList.size() > 0)
882             return true;
883         else
884             return false;
885     }
885
886     /**
887      * Checks if the hand has a wild card.
888      * @return true if there is a wild card, false otherwise.
889      */
890     public boolean hasWildCard34(){
891
892         ArrayList<Integer> rankList = rankPair();
893
894         if (rankList.size() > 0)
895             return true;
896         else
897             return false;
898     }
898
899     /**
900      * Checks if the hand has a wild card.
901      * @return true if there is a wild card, false otherwise.
902      */
903     public boolean hasWildCard35(){
904
905         ArrayList<Integer> rankList = rankPair();
906
907         if (rankList.size() > 0)
908             return true;
909         else
910             return false;
911     }
911
912     /**
913      * Checks if the hand has a wild card.
914      * @return true if there is a wild card, false otherwise.
915      */
916     public boolean hasWildCard36(){
917
918         ArrayList<Integer> rankList = rankPair();
919
920         if (rankList.size() > 0)
921             return true;
922         else
923             return false;
924     }
924
925     /**
926      * Checks if the hand has a wild card.
927      * @return true if there is a wild card, false otherwise.
928      */
929     public boolean hasWildCard37(){
930
931         ArrayList<Integer> rankList = rankPair();
932
933         if (rankList.size() > 0)
934             return true;
935         else
936             return false;
937     }
937
938     /**
939      * Checks if the hand has a wild card.
940      * @return true if there is a wild card, false otherwise.
941      */
942     public boolean hasWildCard38(){
943
944         ArrayList<Integer> rankList = rankPair();
945
946         if (rankList.size() > 0)
947             return true;
948         else
949             return false;
950     }
950
951     /**
952      * Checks if the hand has a wild card.
953      * @return true if there is a wild card, false otherwise.
954      */
955     public boolean hasWildCard39(){
956
957         ArrayList<Integer> rankList = rankPair();
958
959         if (rankList.size() > 0)
960             return true;
961         else
962             return false;
963     }
963
964     /**
965      * Checks if the hand has a wild card.
966      * @return true if there is a wild card, false otherwise.
967      */
968     public boolean hasWildCard40(){
969
970         ArrayList<Integer> rankList = rankPair();
971
972         if (rankList.size() > 0)
973             return true;
974         else
975             return false;
976     }
976
977     /**
978      * Checks if the hand has a wild card.
979      * @return true if there is a wild card, false otherwise.
980      */
981     public boolean hasWildCard41(){
982
983         ArrayList<Integer> rankList = rankPair();
984
985         if (rankList.size() > 0)
986             return true;
987         else
988             return false;
989     }
989
990     /**
991      * Checks if the hand has a wild card.
992      * @return true if there is a wild card, false otherwise.
993      */
994     public boolean hasWildCard42(){
995
996         ArrayList<Integer> rankList = rankPair();
997
998         if (rankList.size() > 0)
999             return true;
1000            else
1001                return false;
1002        }
1002
1003     /**
1004      * Checks if the hand has a wild card.
1005      * @return true if there is a wild card, false otherwise.
1006      */
1007     public boolean hasWildCard43(){
1008
1009         ArrayList<Integer> rankList = rankPair();
1010
1011         if (rankList.size() > 0)
1012             return true;
1013            else
1014                return false;
1015        }
1015
1016     /**
1017      * Checks if the hand has a wild card.
1018      * @return true if there is a wild card, false otherwise.
1019      */
1020     public boolean hasWildCard44(){
1021
1022         ArrayList<Integer> rankList = rankPair();
1023
1024         if (rankList.size() > 0)
1025             return true;
1026            else
1027                return false;
1028        }
1028
1029     /**
1030      * Checks if the hand has a wild card.
1031      * @return true if there is a wild card, false otherwise.
1032      */
1033     public boolean hasWildCard45(){
1034
1035         ArrayList<Integer> rankList = rankPair();
1036
1037         if (rankList.size() > 0)
1038             return true;
1039            else
1040                return false;
1041        }
1041
1042     /**
1043      * Checks if the hand has a wild card.
1044      * @return true if there is a wild card, false otherwise.
1045      */
1046     public boolean hasWildCard46(){
1047
1048         ArrayList<Integer> rankList = rankPair();
1049
1050         if (rankList.size() > 0)
1051             return true;
1052            else
1053                return false;
1054        }
1054
1055     /**
1056      * Checks if the hand has a wild card.
1057      * @return true if there is a wild card, false otherwise.
1058      */
1059     public boolean hasWildCard47(){
1060
1061         ArrayList<Integer> rankList = rankPair();
1062
1063         if (rankList.size() > 0)
1064             return true;
1065            else
1066                return false;
1067        }
1067
1068     /**
1069      * Checks if the hand has a wild card.
1070      * @return true if there is a wild card, false otherwise.
1071      */
1072     public boolean hasWildCard48(){
1073
1074         ArrayList<Integer> rankList = rankPair();
1075
1076         if (rankList.size() > 0)
1077             return true;
1078            else
1079                return false;
1080        }
1080
1081     /**
1082      * Checks if the hand has a wild card.
1083      * @return true if there is a wild card, false otherwise.
1084      */
1085     public boolean hasWildCard49(){
1086
1087         ArrayList<Integer> rankList = rankPair();
1088
1089         if (rankList.size() > 0)
1090             return true;
1091            else
1092                return false;
1093        }
1093
1094     /**
1095      * Checks if the hand has a wild card.
1096      * @return true if there is a wild card, false otherwise.
1097      */
1098     public boolean hasWildCard50(){
1099
1100         ArrayList<Integer> rankList = rankPair();
1101
1102         if (rankList.size() > 0)
1103             return true;
1104            else
1105                return false;
1106        }
1106
1107     /**
1108      * Checks if the hand has a wild card.
1109      * @return true if there is a wild card, false otherwise.
1110      */
1111     public boolean hasWildCard51(){
1112
1113         ArrayList<Integer> rankList = rankPair();
1114
1115         if (rankList.size() > 0)
1116             return true;
1117            else
1118                return false;
1119        }
1119
1120     /**
1121      * Checks if the hand has a wild card.
1122      * @return true if there is a wild card, false otherwise.
1123      */
1124     public boolean hasWildCard52(){
1125
1126         ArrayList<Integer> rankList = rankPair();
1127
1128         if (rankList.size() > 0)
1129             return true;
1130            else
1131                return false;
1132        }
1132
1133     /**
1134      * Checks if the hand has a wild card.
1135      * @return true if there is a wild card, false otherwise.
1136      */
1137     public boolean hasWildCard53(){
1138
1139         ArrayList<Integer> rankList = rankPair();
1140
1141         if (rankList.size() > 0)
1142             return true;
1143            else
1144                return false;
1145        }
1145
1146     /**
1147      * Checks if the hand has a wild card.
1148      * @return true if there is a wild card, false otherwise.
1149      */
1150     public boolean hasWildCard54(){
1151
1152         ArrayList<Integer> rankList = rankPair();
1153
1154         if (rankList.size() > 0)
1155             return true;
1156            else
1157                return false;
1158        }
1158
1159     /**
1160      * Checks if the hand has a wild card.
1161      * @return true if there is a wild card, false otherwise.
1162      */
1163     public boolean hasWildCard55(){
1164
1165         ArrayList<Integer> rankList = rankPair();
1166
1167         if (rankList.size() > 0)
1168             return true;
1169            else
1170                return false;
1171        }
1171
1172     /**
1173      * Checks if the hand has a wild card.
1174      * @return true if there is a wild card, false otherwise.
1175      */
1176     public boolean hasWildCard56(){
1177
1178         ArrayList<Integer> rankList = rankPair();
1179
1180         if (rankList.size() > 0)
1181             return true;
1182            else
1183                return false;
1184        }
1184
1185     /**
1186      * Checks if the hand has a wild card.
1187      * @return true if there is a wild card, false otherwise.
1188      */
1189     public boolean hasWildCard57(){
1190
1191         ArrayList<Integer> rankList = rankPair();
1192
1193         if (rankList.size() > 0)
1194             return true;
1195            else
1196                return false;
1197        }
1197
1198     /**
1199      * Checks if the hand has a wild card.
1200      * @return true if there is a wild card, false otherwise.
1201      */
1202     public boolean hasWildCard58(){
1203
1204         ArrayList<Integer> rankList = rankPair();
1205
1206         if (rankList.size() > 0)
1207             return true;
1208            else
1209                return false;
1210        }
1210
1211     /**
1212      * Checks if the hand has a wild card.
1213      * @return true if there is a wild card, false otherwise.
1214      */
1215     public boolean hasWildCard59(){
1216
1217         ArrayList<Integer> rankList = rankPair();
1218
1219         if (rankList.size() > 0)
1220             return true;
1221            else
1222                return false;
1223        }
1223
1224     /**
1225      * Checks if the hand has a wild card.
1226      * @return true if there is a wild card, false otherwise.
1227      */
1228     public boolean hasWildCard60(){
1229
1230         ArrayList<Integer> rankList = rankPair();
1231
1232         if (rankList.size() > 0)
1233             return true;
1234            else
1235                return false;
1236        }
1236
1237     /**
1238      * Checks if the hand has a wild card.
1239      * @return true if there is a wild card, false otherwise.
1240      */
1241     public boolean hasWildCard61(){
1242
1243         ArrayList<Integer> rankList = rankPair();
1244
1245         if (rankList.size() > 0)
1246             return true;
1247            else
1248                return false;
1249        }
1249
1250     /**
1251      * Checks if the hand has a wild card.
1252      * @return true if there is a wild card, false otherwise.
1253      */
1254     public boolean hasWildCard62(){
1255
1256         ArrayList<Integer> rankList = rankPair();
1257
1258         if (rankList.size() > 0)
1259             return true;
1260            else
1261                return false;
1262        }
1262
1263     /**
1264      * Checks if the hand has a wild card.
1265      * @return true if there is a wild card, false otherwise.
1266      */
1267     public boolean hasWildCard63(){
1268
1269         ArrayList<Integer> rankList = rankPair();
1270
1271         if (rankList.size() > 0)
1272             return true;
1273            else
1274                return false;
1275        }
1275
1276     /**
1277      * Checks if the hand has a wild card.
1278      * @return true if there is a wild card, false otherwise.
1279      */
1280     public boolean hasWildCard64(){
1281
1282         ArrayList<Integer> rankList = rankPair();
1283
1284         if (rankList.size() > 0)
1285             return true;
1286            else
1287                return false;
1288        }
1288
1289     /**
1290      * Checks if the hand has a wild card.
1291      * @return true if there is a wild card, false otherwise.
1292      */
1293     public boolean hasWildCard65(){
1294
1295         ArrayList<Integer> rankList = rankPair();
1296
1297         if (rankList.size() > 0)
1298             return true;
1299            else
1300                return false;
1301        }
1301
1302     /**
1303      * Checks if the hand has a wild card.
1304      * @return true if there is a wild card, false otherwise.
1305      */
1306     public boolean hasWildCard66(){
1307
1308         ArrayList<Integer> rankList = rankPair();
1309
1310         if (rankList.size() > 0)
1311             return true;
1312            else
1313                return false;
1314        }
1314
1315     /**
1316      * Checks if the hand has a wild card.
1317      * @return true if there is a wild card, false otherwise.
1318      */
1319     public boolean hasWildCard67(){
1320
1321         ArrayList<Integer> rankList = rankPair();
1322
1323         if (rankList.size() > 0)
1324             return true;
1325            else
1326                return false;
1327        }
1327
1328     /**
1329      * Checks if the hand has a wild card.
1330      * @return true if there is a wild card, false otherwise.
1331      */
1332     public boolean hasWildCard68(){
1333
1334         ArrayList<Integer> rankList = rankPair();
1335
1336         if (rankList.size() > 0)
1337             return true;
1338            else
1339                return false;
1340        }
1340
1341     /**
1342      * Checks if the hand has a wild card.
1343      * @return true if there is a wild card, false otherwise.
1344      */
1345     public boolean hasWildCard69(){
1346
1347         ArrayList<Integer> rankList = rankPair();
1348
1349         if (rankList.size() > 0)
1350             return true;
1351            else
1352                return false;
1353        }
1353
1354     /**
1355      * Checks if the hand has a wild card.
1356      * @return true if there is a wild card, false otherwise.
1357      */
1358     public boolean hasWildCard70(){
1359
1360         ArrayList<Integer> rankList = rankPair();
1361
1362         if (rankList.size() > 0)
1363             return true;
1364            else
1365                return false;
1366        }
1366
1367     /**
1368      * Checks if the hand has a wild card.
1369      * @return true if there is a wild card, false otherwise.
1370      */
1371     public boolean hasWildCard71(){
1372
1373         ArrayList<Integer> rankList = rankPair();
1374
1375         if (rankList.size() > 0)
1376             return true;
1377            else
1378                return false;
1379        }
1379
1380     /**
1381      * Checks if the hand has a wild card.
1382      * @return true if there is a wild card, false otherwise.
1383      */
1384     public boolean hasWildCard72(){
1385
1386         ArrayList<Integer> rankList = rankPair();
1387
1388         if (rankList.size() > 0)
1389             return true;
1390            else
1391                return false;
1392        }
1392
1393     /**
1394      * Checks if the hand has a wild card.
1395      * @return true if there is a wild card, false otherwise.
1396      */
1397     public boolean hasWildCard73(){
1398
1399         ArrayList<Integer> rankList = rankPair();
1400
1401         if (rankList.size() > 0)
1402             return true;
1403            else
1404                return false;
1405        }
1405
1406     /**
1407      * Checks if the hand has a wild card.
1408      * @return true if there is a wild card, false otherwise.
1409      */
1410     public boolean hasWildCard74(){
1411
1412         ArrayList<Integer> rankList = rankPair();
1413
1414         if (rankList.size
```

```

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 efficiency 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
344      * representing the hand's ranks.
345     */
346     private ArrayList<Integer> rankTwoPair(){
347
348         ArrayList<Integer> hand = new ArrayList<>();
349         hand = this.ranksOfHand(); // our
350         unmodified hand
351
352         ArrayList<Integer> rankList = new ArrayList
353         <>(); // our new list of (int) ranks that we want to
354         return
355         int pairRank1 = 0;
356         int pairRank2 = 0;
357
358         int pairRank3 = 0;
359         int pairRank4 = 0;
360
361         for (int i = 0; i < MAX_CARDS_IN_HAND - 1; i
362         ++) {
363             for (int j = (i + 1); j <
364             MAX_CARDS_IN_HAND; j++) {
365                 if (this.ranksOfHand().get(i).equals
366                 (this.ranksOfHand().get(j))) { //pair detection
367                     pairRank1 = hand.get(i);
368                     pairRank2 = hand.get(j);
369                 }
370             }
371         }
372
373         rankList.add(pairRank1);
374         hand.remove((Object) pairRank1);
375         hand.remove((Object) pairRank2);
376
377         for (int i = 0; i < (hand.size() - 1); i++){
378             for (int j = (i + 1); j < hand.size(); j
379             ++){

```

```

372                     if (hand.get(i) == hand.get(j)){ //  

373                         pair detection  

374                         pairRank3 = hand.get(i);  

375                         pairRank4 = hand.get(j);  

376                     }  

377                 }  

378             }  

379             if (rankList.get(0) > pairRank3){ // orders  

380                 the two pair ranks  

381                     rankList.add(pairRank3);  

382                 }  

383                 else {  

384                     rankList.add(0, pairRank3);  

385                 }  

386                 hand.remove((Object) pairRank3);  

387                 hand.remove((Object) pairRank4);  

388                 rankList.add(hand.get(0));  

389             }  

390             return rankList;  

391         }  

392         /**  

393          * Only goes off when the hand is a 'HighCard'.  

394          * @return Returns the highest Rank of the hand.  

395          */  

396         private ArrayList<Integer> rankHighCard(){  

397             ArrayList<Card> hand = new ArrayList<>();  

398             hand = this.cardList;  

399             ArrayList<Integer> ranksOfHighCard = new  

400             ArrayList<>();  

401             for (int rank = HIGHEST_RANK; rank >= 0;  

402             rank--){  

403                 for (int card = 0; card<hand.size();  

404                 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() ==  
412             MAX_CARDS_IN_HAND){ // for efficiency purposes  
413             return ranksOfHighCard;  
414         }  
415         return null; // here for syntax purposes,  
416         will not go off.  
417     }  
418 }  
419
```

```
1 package proj4;
2
3 public class CardTester {
4     public static void main(String[] args) {
5         Testing.startTests();
6         testToStringINT();
7         testToStringSTR();
8         testGetRank1();
9         testGetRank2();
10        testGetSuit1();
11        testGetSuit2();
12        Testing.finishTests();
13    }
14
15    public static void testToStringINT(){
16        Card kingDiamond = new Card(12,3);
17
18        Testing.assertEquals("Tests toString for a
19        Queen of diamonds (INT)"
20        , "Queen of Diamonds"
21        , kingDiamond.toString());
22
23    public static void testToStringSTR(){
24        Card threeDiamond = new Card("3","Clubs");
25
26        Testing.assertEquals("Tests toString for a 3
27        of Clubs (STR)"
28        , "3 of Clubs"
29        , threeDiamond.toString());
30
31    public static void testGetRank1(){
32        Card threeDiamond = new Card("three",
33        "diamonds");
34
35        Testing.assertEquals("Tests toString for the
36        rank of a 3 of diamonds(STR)"
37        , 3
38        , threeDiamond.getRank());
39    }
```

```
38
39     public static void testGetRank2(){
40         Card threeDiamond = new Card("jack", "Spades"
41     );
42
43         Testing.assertEquals("Tests toString for the
44             rank of a Jack of Spades(STR)"
45             , 11
46             , threeDiamond.getRank());
47
48
49     public static void testGetSuit1(){
50         Card eightClubs = new Card(8, 2);
51
52         Testing.assertEquals("Tests toString for the
53             suit of an Eight of Clubs(STR)"
54             , "Clubs"
55             , eightClubs.getSuit());
56
57
58     public static void testGetSuit2(){
59         Card eightClubs = new Card(14, 1);
60
61         Testing.assertEquals("Tests toString for the
62             suit of an Ace of Hearts(STR)"
63             , "Hearts"
64             , eightClubs.getSuit());
65     }
66 }
```

```

1 package proj4;
2
3 public class DeckTester {
4
5     //Deck is constructed by INTS RANK 2-14, SUIT 0-3
6     //for each.
7     // Alternatively, you can use STRINGS "#" and "
8     //suit"
9
10    public static void main(String[] args) {
11        Testing.startTests();
12        testToString();
13        testToStringLast3();
14        testShuffle();
15        testIsEmpty();
16        testGather();
17        testSize();
18        Testing.finishTests();
19    }
20
21    public static void testToString(){
22        Deck deck1 = new Deck();
23        final int CARDS_TO_SKIP = 0;
24        for (int x = 0; x<CARDS_TO_SKIP; x++){
25            deck1.deal();
26        }
27        Testing.assertEquals("Tests a printout of the
28        entire deck"
29                    , "The entire deck, Spades->Clubs->
30        Hearts->Diamonds for ranks 2-Ace"
31                    , deck1.toString());
32    }
33
34    public static void testToStringLast3(){
35        Deck deck1 = new Deck();
36        final int CARDS_TO_SKIP = 49;
37        for (int x = 0; x<CARDS_TO_SKIP; x++){
38            deck1.deal();
39        }
40        Testing.assertEquals("Tests a printout of the
41        last 3 cards of the deck"
42                    , "[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"  
                , "52 cards in random order"  
                , 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"  
                , true  
                , 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"  
                , 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
84                             remaining cards"
85                             , 22
86                             , deck1.size());
87
88 }
89
```

```
1 package proj4;
2
3 import java.util.ArrayList;
4
5 public class StudPokerHand {
6
7     private final int MAX_CARDS_IN_STUD = 2;
8     private final int MAX_CARDS_IN_HAND = 5;
9
10    private ArrayList<Card> cardList;
11    private CommunityCardSet cCards;
12
13    /**
14     * Constructor.
15     * @param cc The set of Community Cards currently
16     * in play.
17     * @param cardList the StudPokerHand we are
18     * constructing.
19     */
20    public StudPokerHand(CommunityCardSet cc,
21        ArrayList<Card> cardList){
22        this.cardList = cardList;
23        this.cCards = cc;
24    }
25
26    /**
27     * Adds a card to the StudHand.
28     * If the hand is full, does nothing.
29     * @param CardObject The card that will be added.
30     */
31    public void addCard(Card CardObject){
32        if (cardList.size() <= MAX_CARDS_IN_STUD){
33            cardList.add(CardObject);
34        }
35        else {
36            return;
37        }
38    }
39
40    /**
41     * Returns a card from the Stud.
42     */
43}
```

```

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

```

```

71      *
72      * @param other The hand to compare this hand to
73      * @return a negative number if this is worth
74      LESS than other, zero
75      * if they are worth the SAME, and a positive
76      number if this is worth
77      * MORE than other
78      */
79      public int compareTo(StudPokerHand other){
80
81          PokerHand hand1 = this.getBestFiveCardHand
82          ();
83          PokerHand hand2 = other.getBestFiveCardHand
84          ();
85
86          /**
87          * Determines every possible hand that can be
88          created from a stud and the
89          * available community cards, and returns a list
90          of them.
91          * @return An ArrayList of all Pokerhands made
92          from the 7 cards available.
93          */
94          private ArrayList<PokerHand> getAllFiveCardHands
95          (){
96              ArrayList<PokerHand> handList = new
97              ArrayList<>();
98
99              for (int i = 0; i < 6; i++){ // elimination
loop
100                  for (int j = (i + 1); j < 6; j++){
101
102                      ArrayList<Card> sevenList = new
103                      ArrayList<>(); //our seven cards
104
105                      for (int index = 0; index <
MAX_CARDS_IN_HAND; index++){

```

```

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

```

```
1 package proj4;
2
3 /*
4 I affirm that I have carried out the attached
5 academic endeavors with full academic honesty, in
6 accordance with the Union College Honor Code and the
7 course syllabus.
8 */
9
10 public class PokerHandTester {
11
12     public static void main(String[] args) {
13         //calls go here
14         //CHECK VERBOSITY BOOLEAN IN TESTING//
15
16         Testing.startTests();
17         testCompareToHC1();
18         testCompareTo2Flushes1();
19         testCompareTo2Pairs1();
20         testCompareTo2TwoPairs1();
21         testCompareTo2Pairs2();
22         testCompareTo2twoPairs2();
23         testGetIthCard();
24         testToString();
25         testAddCard();
26         Testing.finishTests();
27
28     }
29     //methods go here
30
31     public static void testCompareToHC1(){
32         Card c1 = new Card("11","Hearts");
33         Card c2 = new Card("7","Hearts");
34         Card c3 = new Card("14","Hearts");
35         Card c4 = new Card("9","Diamonds");
36         Card c5 = new Card("3","Hearts");
37         ArrayList<Card> array1 = new ArrayList<>();
38         array1.add(c1);
39         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
91             the 2nd highest card"
92             , -1
93             , hand1.compareTo(hand2));
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
124             pair of queens vs a pair of jacks"
125             , 1
126             , hand1.compareTo(hand2));
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
156             twopair of 10s vs a twopair of 8s"
157             , 1
158             , hand1.compareTo(hand2));
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
220             that break tie with their last chaser"
221             , -1
222             , hand1.compareTo(hand2));
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
239             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
258                             pokerhand"
259                             , "[10 of Hearts, 10 of Diamonds, 8
260                             of Hearts, 8 of Hearts, 3 of Hearts]"
261                             , hand1.toString());
262     }
263
264     public static void testAddCard(){
265         Card d1 = new Card(14,3);
266         Card d2 = new Card(12,3);
267         Card d3 = new Card(9,3);
268         Card d4 = new Card(9,3);
269         Card d5 = new Card(13,3);
270         ArrayList<Card> array2 = new ArrayList<>();
271         PokerHand hand2 = new PokerHand(array2);
272         hand2.addCard(d1);
273         hand2.addCard(d2);
274         hand2.addCard(d3);
275         hand2.addCard(d4);
276         hand2.addCard(d5);
277
278         Testing.assertEquals("Tests 2 flushes for
279                             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
```

```

1 package proj4;
2
3 import java.util.ArrayList;
4
5 public class CommunityCardSet {
6
7     private final int MAX_CARDS_IN_HAND = 5;
8     private ArrayList<Card> cCards;
9
10    /**
11     * Constructor.
12     * @param cCards An arrayList of community cards
13     * any hand can make use of.
14     */
15    public CommunityCardSet(ArrayList<Card> cCards){
16        this.cCards = cCards;
17    }
18
19    /**
20     * Adds a card to the cCard Set.
21     * If the set is full, does nothing.
22     * @param CardObject The card that will be added.
23     */
24    public void addCard(Card CardObject){
25
26        if (cCards.size() <= MAX_CARDS_IN_HAND){
27            cCards.add(CardObject);
28        }
29        else {
30            return;
31        }
32    }
33
34    /**
35     * Returns a card from the set.
36     * @param index The index of the card we are
37     * getting.
38     * @return returns the card at the specified
39     * index.
40     * If the index is invalid, returns nothing at
41     * all.

```

```
39     */
40     public Card get_ith_card(int index){
41
42         if ((index >= 0) || (index <
43             MAX_CARDS_IN_HAND)){
44             return cCards.get(index);
45         }
46         else {
47             return null;
48         }
49
50     /**
51      * Returns all the cards in the communityCard set
52
53      * @return Returns an easily readable string of
54      * all the cards in the set.
55      */
56     public String toString(){
57
58         ArrayList<Card> handList = new ArrayList<>();
59         for (int card=0; card<cCards.size(); card++){
60             handList.add( cCards.get(card) );
61         }
62
63         return handList.toString();
64     }
65 }
```

```
1 package proj4;
2
3 import java.util.ArrayList;
4
5 public class StudPokerHandTester {
6
7     public static void main(String[] args) {
8         Testing.startTests();
9         testToString();
10        testAddCard();
11        testGetIthCard();
12        Testing.finishTests();
13    }
14
15    public static void testToString(){
16        Card c1 = new Card("11","Hearts");
17        Card c2 = new Card("7","Hearts");
18        Card c3 = new Card("14","Hearts");
19        Card c4 = new Card("9","Diamonds");
20        Card c5 = new Card("3","Hearts");
21
22        Card d1 = new Card(14,3);
23        Card d2 = new Card(12,3);
24
25        ArrayList<Card> array1 = new ArrayList<>();
26        CommunityCardSet cCards = new
27            CommunityCardSet(array1);
28        ArrayList<Card> array2 = new ArrayList<>();
29        array2.add(d1);
30        array2.add(d2);
31
32        StudPokerHand hand2 = new StudPokerHand(
33            cCards, array2);
34
35        Testing.assertEquals("Tests toString of the
36            stud"
37                , "[Ace of Diamonds, Queen of
38            Diamonds]"
39                , hand2.toString());
40    }
41}
```

```

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
50             CommunityCardSet(array1);
51         ArrayList<Card> array2 = new ArrayList<>();
52
53         StudPokerHand hand2 = new StudPokerHand(
54             cCards, array2);
55         hand2.addCard(d1);
56         hand2.addCard(d2);
57
58         Testing.assertEquals("Tests addCard of the
59             stud"
60                     , "[4 of Spades, Jack of Clubs]"
61                     , hand2.toString());
62
63     public static void testGetIthCard(){
64         Card c1 = new Card("11","Hearts");
65         Card c2 = new Card("7","Hearts");
66         Card c3 = new Card("14","Hearts");
67         Card c4 = new Card("9","Diamonds");
68         Card c5 = new Card("3","Hearts");
69
70         Card d1 = new Card(14,3);
71         Card d2 = new Card(12,3);
72
73         ArrayList<Card> array1 = new ArrayList<>();
74         CommunityCardSet cCards = new
75             CommunityCardSet(array1);
76         ArrayList<Card> array2 = new ArrayList<>();
77         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
```

```
1 package proj4;
2
3 import java.util.ArrayList;
4
5 public class CommunityCardSetTester {
6
7     public static void main(String[] args) {
8
9         Testing.startTests();
10        testToString();
11        testAddCard();
12        testGetIthCard();
13        Testing.finishTests();
14    }
15
16    public static void testToString(){
17        Card c1 = new Card("11","Hearts");
18        Card c2 = new Card("7","Hearts");
19        Card c3 = new Card("14","Hearts");
20        Card c4 = new Card("9","Diamonds");
21        Card c5 = new Card("3","Hearts");
22        ArrayList<Card> array1 = new ArrayList<>();
23        array1.add(c1);
24        array1.add(c2);
25        array1.add(c3);
26        array1.add(c4);
27        array1.add(c5);
28        PokerHand hand1 = new PokerHand(array1);
29
30        Testing.assertEquals("Tests for
31            communitycardsets toString"
32                    , "[Jack of Hearts, 7 of Hearts, Ace
33            of Hearts, 9 of Diamonds, 3 of Hearts]"
34                    , hand1.toString());
35    }
36
37    public static void testAddCard(){
38        Card d1 = new Card(2,0);
39        Card d2 = new Card(12,3);
40        Card d3 = new Card(10,2);
41        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
50                             2nd highest card"
50                             , "[2 of Spades, Queen of Diamonds,
50                             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
69                             the 2nd indexed card of the pokerhand"
70                             , "8 of Hearts"
71                             , hand1.get_ith_card(2));
72     }
73 }
```