

```

1  package proj3; // do not erase. Gradescope expects this.
2
3  public class Card {
4
5
6      private int cardRank; // Rank of the card
7      private String cardSuit; // Suit of the card
8
9      /**
10       * Card Object Constructor
11       * @param rank rank of the card
12       * @param suit suit of the card
13       */
14     public Card(int rank, String suit){
15
16         cardRank = rank;
17         cardSuit = suit;
18
19     }
20
21
22     /**
23      * getter for the rank of the card
24      * @return int: rank
25      */
26     public int getCardRank(){
27         return cardRank;
28     }
29
30     /**
31      * Getter for the suit of the card
32      *
33      * @return String: suit
34      */
35     public String getCardSuit(){
36         return cardSuit;
37     }
38
39     /**
40      * format the Card as a string for printing and such
41      *
42      * @return String of the card
43      */
44     public @Override String toString(){
45
46         String rankHolder = Integer.toString(cardRank);
47         if(cardRank == 11){
48             rankHolder = "Jack";
49         }
50         if(cardRank == 12){
51             rankHolder = "Queen";
52         }

```

```
53         if(cardRank == 13){
54             rankHolder = "King";
55         }
56         if(cardRank == 14){
57             rankHolder = "Ace";
58         }
59
60         String cardString = "[ " + cardRank + " of " +
        cardSuit + " ]";
61         return cardString;
62
63     }
64
65 }
66
```

```

1  package proj3; // do not erase. Gradescope expects this.
2
3  import java.util.*;
4  import java.util.concurrent.ThreadLocalRandom;
5
6  /**
7   * Class for modeling a Deck of Cards
8   */
9  public class Deck {
10
11     private int DECK_SIZE = 52; // standard deck size
12     private ArrayList<Card> cardList = new ArrayList<Card>
13     >(DECK_SIZE); // ArrayList of cards
14     private int nextToDeal = 0; // initialized variable of
15     index we are drawing next card from
16
17     /**
18     * Constructor for a Deck object
19     */
20     public Deck() {
21         generateDeck(); // generates the deck
22     }
23
24     /**
25     * Generates a standard 52 card deck
26     * ranks: 2 to 14
27     * suits: Hearts, Diamonds, Spades, Clubs
28     */
29     private void generateDeck() {
30         for (int i = 0; i < 4; i++) {
31             for (int j = 2; j <= 14; j++) {
32                 Card card = null;
33                 if (i == 0) {
34                     card = new Card(j, "Hearts");
35                 }
36                 if (i == 1) {
37                     card = new Card(j, "Diamonds");
38                 }
39                 if (i == 2) {
40                     card = new Card(j, "Clubs");
41                 }
42                 if (i == 3) {
43                     card = new Card(j, "Spades");
44                 }
45                 cardList.add(card);
46             }
47         }
48
49     /**
50     * shuffles the deck of cards

```

```

51     */
52     public void shuffle(){
53
54         for(int i = nextToDeal; i < cardList.size(); i
55         ++){
56
57             Card currentCard = cardList.get(i);
58
59             int random = ThreadLocalRandom.current().
60             nextInt(i, cardList.size());
61
62             Card swapCard = cardList.get(random);
63
64             cardList.set(i, swapCard);
65             cardList.set(random, currentCard);
66         }
67     }
68
69     /**
70      * deals the next card in the deck
71      * efficiency: O(1)
72      * @return previously undelt Card
73      */
74     public Card deal(){
75         if(nextToDeal == this.size()){
76             return null;
77         }
78         else {
79             Card topCard = cardList.get(nextToDeal);
80             nextToDeal++;
81             return topCard;
82         }
83     }
84
85     /**
86      * return size of the deck; number of undelt cards
87      * @return int value of number of cards
88      */
89     public int size(){
90         int deckSize;
91         deckSize = cardList.size() - nextToDeal;
92         return deckSize;
93     }
94
95     /**
96      * reset the next card to deal to the first in the
97      deck
98      */
99     public void gather(){
100         nextToDeal = 0;

```

```
100     }
101
102     /**
103      * format the deck as a string for printing and such
104      * @return String
105      */
106     public @Override String toString(){
107         String str = "";
108         for(int i = nextToDeal; i < cardList.size(); i++){
109             str += cardList.get(i).toString();
110             str += "\n";
111         }
112         return str;
113     }
114
115 }
116
```

```

1  package proj3;
2
3  /**
4   * This class contains a collection of methods that help
   * with testing. All methods
5   * here are static so there's no need to construct a
   * Testing object. Just call them
6   * with the class name like so:
7   * 

<p></p>


8   * Testing.assertEquals("test description", expected
   * , actual)</code>
9   *
10  * @author Kristina Striegnitz, Aaron Cass, Chris
   * Fernandes
11  * @version 5/28/18
12  */
13  public class Testing {
14
15      private static boolean VERBOSE = false;
16      private static int numTests;
17      private static int numFails;
18
19      /**
20       * Toggles between a lot of output and little output.
21       *
22       * @param verbose
23       * If verbose is true, then complete
   * information is printed,
24       * whether the tests passes or fails. If
   * verbose is false, only
25       * failures are printed.
26       */
27      public static void setVerbose(boolean verbose)
28      {
29          VERBOSE = verbose;
30      }
31
32      /**
33       * Each of the assertEquals methods tests whether the
   * actual
34       * result equals the expected result. If it does, then
   * the test
35       * passes, otherwise it fails.
36       *
37       * The only difference between these methods is the
   * types of the
38       * parameters.
39       *
40       * All take a String message and two values of some
   * other type to
41       * compare:
42       */

```

```

43      * @param message
44      *           a message or description of the test
45      * @param expected
46      *           the correct, or expected, value
47      * @param actual
48      *           the actual value
49      */
50      public static void assertEquals(String message,
boolean expected,
51                                     boolean actual)
52      {
53          printTestCaseInfo(message, "\"" + expected, "\"" +
actual);
54          if (expected == actual) {
55              pass();
56          } else {
57              fail(message);
58          }
59      }
60
61      public static void assertEquals(String message, int
expected, int actual)
62      {
63          printTestCaseInfo(message, "\"" + expected, "\"" +
actual);
64          if (expected == actual) {
65              pass();
66          } else {
67              fail(message);
68          }
69      }
70
71      public static void assertEquals(String message, Object
expected,
72                                     Object actual)
73      {
74          String expectedString = "<<null>>";
75          String actualString = "<<null>>";
76          if (expected != null) {
77              expectedString = expected.toString();
78          }
79          if (actual != null) {
80              actualString = actual.toString();
81          }
82          printTestCaseInfo(message, expectedString,
actualString);
83
84          if (expected == null) {
85              if (actual == null) {
86                  pass();
87              } else {
88                  fail(message);

```

```

89         }
90     } else if (expected.equals(actual)) {
91         pass();
92     } else {
93         fail(message);
94     }
95 }
96
97 /**
98  * Asserts that a given boolean must be true. The
test fails if
99  * the boolean is not true.
100  *
101  * @param message The test message
102  * @param actual The boolean value asserted to be
true.
103  */
104 public static void assertTrue(String message, boolean
actual)
105 {
106     assertEquals(message, true, actual);
107 }
108
109 /**
110  * Asserts that a given boolean must be false. The
test fails if
111  * the boolean is not false (i.e. if it is true).
112  *
113  * @param message The test message
114  * @param actual The boolean value asserted to be
false.
115  */
116 public static void assertFalse(String message,
boolean actual)
117 {
118     assertEquals(message, false, actual);
119 }
120
121 private static void printTestCaseInfo(String message
, String expected,
122                                     String actual)
123 {
124     if (VERBOSE) {
125         System.out.println(message + ":");
126         System.out.println("expected: " + expected);
127         System.out.println("actual:   " + actual);
128     }
129 }
130
131 private static void pass()
132 {
133     numTests++;

```



```

134
135         if (VERBOSE) {
136             System.out.println("--PASS--");
137             System.out.println();
138         }
139     }
140
141     private static void fail(String description)
142     {
143         numTests++;
144         numFails++;
145
146         if (!VERBOSE) {
147             System.out.print(description + " ");
148         }
149         System.out.println("--FAIL--");
150         System.out.println();
151     }
152
153     /**
154      * Prints a header for a section of tests.
155      *
156      * @param sectionTitle The header that should be
157      * printed.
158      */
159     public static void testSection(String sectionTitle)
160     {
161         if (VERBOSE) {
162             int dashCount = sectionTitle.length();
163             System.out.println(sectionTitle);
164             for (int i = 0; i < dashCount; i++) {
165                 System.out.print("-");
166             }
167             System.out.println();
168             System.out.println();
169         }
170     }
171
172     /**
173      * Initializes the test suite. Should be called
174      * before running any
175      * tests, so that passes and fails are correctly
176      * tallied.
177      */
178     public static void startTests()
179     {
180         System.out.println("Starting Tests");
181         System.out.println();
182         numTests = 0;
183         numFails = 0;
184     }

```

```
183     /**
184      * Prints out summary data at end of tests.  Should
185      * be called
186      * after all the tests have run.
187      */
187     public static void finishTests()
188     {
189         System.out.println("=====");
190         System.out.println("Tests Complete");
191         System.out.println("=====");
192         int numPasses = numTests - numFails;
193
194         System.out.print(numPasses + "/" + numTests + "
195     PASS ");
196         System.out.printf("(pass rate: %.1f%s)\n",
197             100 * ((double) numPasses) /
198             numTests,
199             "%");
200
201         System.out.print(numFails + "/" + numTests + "
202     FAIL ");
203         System.out.printf("(fail rate: %.1f%s)\n",
204             100 * ((double) numFails) /
205             numTests,
206             "%");
207     }
208 }
```

```

1  package proj3; // do not erase. Gradescope expects this.
2
3  import java.util.*;
4
5  public class PokerHand {
6
7      private int MAX_HAND_SIZE = 5;
8      private ArrayList<Card> cardsInHand; //all the cards
        in the hand
9
10     private ArrayList<Integer> allRanks = new ArrayList<
        Integer>(); //for seperating out the ranks of the cards
11     private ArrayList<String> allSuits = new ArrayList<
        String>(); //for seperating out the suits
12
13     private ArrayList<Integer> pairRanks = new ArrayList<
        Integer>(); //for seperating out the pairs
14     private ArrayList<Integer> highcardRanks = new
        ArrayList<Integer>(); //for seperating out the non-pairs
15
16     private TreeMap<Integer, Integer> rankOccurances = new
        TreeMap<Integer, Integer>(Collections.reverseOrder()); //
        experimenting with a new data structure, makes sorting
        fuctions simplier
17
18     /**
19      * A Construcuter for a PokerHand Object
20      *
21      * @param cardList cards that will make up the
        PokerHand
22      */
23     public PokerHand(ArrayList<Card> cardList) {
24         cardsInHand = cardList;
25     }
26
27     /**
28      * add a card to the Poker Hand if there are less than
        5 cards in the hand
29      * otherwise do nothing
30      *
31      * @param card card being added to the PokerHand
32      */
33     public void addCard(Card card) {
34
35         if (cardsInHand.size() < MAX_HAND_SIZE) {
36             cardsInHand.add(card);
37         }
38     }
39
40     /**
41      * return the card in the pokerHand at the given index
42      *

```

```

43      * @param index index of card being retrieved
44      * @return Card
45      */
46      public Card get_ith_card(int index) {
47          if (index >= 0 && index < cardsInHand.size() - 1
48      ) {
49              return cardsInHand.get(index);
50          } else {
51              return null;
52          }
53      }
54      /**
55       * override the toString function to turn a PokerHand
56       into a properly formatted string
57       * @return String
58       */
59      public @Override String toString() {
60          String str = "";
61          for (int i = 0; i < cardsInHand.size(); i++) {
62              str += cardsInHand.get(i).toString();
63              str += "\n";
64          }
65          return str;
66      }
67      /**
68       * Determines how this hand compares to another hand,
69       returns
70       * positive, negative, or zero depending on the
71       comparison.
72       * @param other The hand to compare this hand to
73       * @return a negative number if this is worth LESS
74       than other, zero
75       * if they are worth the SAME, and a positive number
76       if this is worth
77       * MORE than other
78       */
79      public int compareTo(PokerHand other) {
80          //organize the hands and determine their types
81          //this hand
82          this.getHandData();
83          Integer hand1Type = this.handType();
84          //otherhand
85          other.getHandData();
86          Integer hand2Type = other.handType();
87          int handTypeComparison = hand1Type.compareTo(
88          hand2Type);

```

```

88         if(handTypeComparison != 0){
89             return handTypeComparison;
90         }
91
92         else { //if hands are of the same type...
93             if(!pairRanks.isEmpty()){ //if there are
pairs to compare...
94                 int pairCompare = this.tieBreaker(this.
pairRanks, other.pairRanks); //compare them
95                 if(pairCompare == 0){ //if the pair
values are equal
96                     return this.tieBreaker(this.
highcardRanks, other.highcardRanks); // return the
highcard comparison
97                 }
98                 else{ //otherwise return the pair
comparison
99                     return pairCompare;
100                 }
101             }
102         }
103         else{ //if there are no pairs to compare,
just return the highcard comparison
104             return this.tieBreaker(this.highcardRanks
, other.highcardRanks);
105         }
106     }
107 }
108
109 /**
110  * Determing the type of the hand. Flush, 2pair,
1pair, or highcard
111  *
112  * @return Integer : 4 if flush, 3 if 2pair, 2 if
1pair, 1 if highcard
113  */
114 private Integer handType() {
115
116     boolean isFlush = flushCheck();
117     if(isFlush) {
118         return 4; //FLUSH
119     }
120     if (pairRanks.size() == 2) { // if you have 2
pairs
121         return 3; //2Pair
122     }
123     if (pairRanks.size() == 1) { //if you have 1 pair
124         return 2; //1 PAIR
125     }
126     else { //If its not a flush, 2pair, or 1pair it
has to be....
127         return 1; //HIGHCARD

```

```

128         }
129
130     }
131
132     /**
133      * Breaks up all the hand data into managable chunks:
134      * seperates suits and ranks into their own
135      ArrayLists,
136      * creates a Treemap of the ranks and their # of
137      occurrences,
138      * uses the Treemap to fill ArrayLists with pair
139      values, and non-pair(highcard) values
140      */
141     private void getHandData() {
142
143         int size = cardsInHand.size(); //size of the hand
144
145         for (int i = 0; i < size; i++) {
146             Card currentCard = cardsInHand.get(i); //
147             //current card we are pulling data from
148             allSuits.add(currentCard.getCardSuit()); //
149             //adding the current cards suit to the suit array
150             allRanks.add(currentCard.getCardRank()); //
151             //adding the current cards rank to the rank array
152         }
153
154         //sort allRanks into a TreeMap with Key = Rank &
155         //Value = instances of the rank
156         for (Integer i : allRanks) {
157             Integer j = rankOccurrences.get(i);
158             rankOccurrences.put(i, (j == null) ? 1 : j + 1
159 );
160         }
161
162         //sort rankOccurrences by pairs and non-pairs(
163         highcards)
164         for (Integer key : rankOccurrences.keySet()) {
165             if (rankOccurrences.get(key) == 4) { //2pair
166                 pairRanks.add(key);
167                 pairRanks.add(key);
168             }
169             if (rankOccurrences.get(key) == 2 ||
170 rankOccurrences.get(key) == 3) {
171                 pairRanks.add(key);
172             }
173             else {
174                 highcardRanks.add(key);
175             }
176         }
177     }
178 }
179

```

```

170     /**
171      * compares two ArrayLists of ranks and determines
172      * which has the first instance of a greater value
173      *
174      * @param theseRanks ArrayList of ranks from this
175      * hand
176      * @param otherRanks ArrayList of ranks from other
177      * hand
178      * @return int 1 if theseRanks is greater, -1 if
179      * otherRanks is greater, 0 if all ranks are the same
180      */
181     private int tieBreaker(ArrayList<Integer> theseRanks
182     , ArrayList<Integer> otherRanks) {
183
184         //compare each rank
185         for (int i = 0; i < theseRanks.size(); i++) {
186             int currentCompare = theseRanks.get(i).
187             compareTo(otherRanks.get(i)); //compare current index
188             if (currentCompare != 0) { //if the current
189             index ranks are different...
190                 return currentCompare; //return the
191             comparison
192         }
193     }
194     return 0; //you make it through all ranks and
195     they are all the same
196 }
197
198 /**
199  * checks if the hand is a flush
200  *
201  * @return true if hand is a flush, false if not
202  */
203 private boolean flushCheck(){
204
205     //checking for a flush
206     String checkSuit = allSuits.get(1); //a suit
207     present in the hand
208     if(Collections.frequency(allSuits, checkSuit) ==
209     cardsInHand.size()){
210         return true;
211     }
212     else{
213         return false;
214     }
215 }
216 }

```

```

1  package proj3;
2  import org.junit.Test;
3
4  import java.util.ArrayList;
5  import java.util.Arrays;
6  import java.util.concurrent.ThreadLocalRandom;
7
8  /**
9   * Author: Ian Sulley
10  *
11  * Honor Code: I affirm that I have carried out the
    attached academic endeavors
12  * with full academic honesty, in accordance with the
    Union College Honor Code
13  * and the course syllabus
14  */
15
16 /**
17  * Testing Class for PokerHand compareTo()
18  */
19 public class PokerComparisonTests {
20
21     public static void main(String[] args) {
22
23         test_all();
24     }
25
26     //#####TESTS#####
27     public static void test_all(){
28         Testing.startTests();
29         test_all_flushes();
30         test_all_two_pair();
31         test_all_pair();
32         Testing.finishTests();
33     }
34
35
36
37     //#####FLUSH TESTS#####
38
39     public static void test_all_flushes() {
40         Testing.startTests();
41         compare_flushes1();
42         compare_flushes2();
43         compare_flushes_tie();
44         compare_flush_2pair();
45         compare_flush_pair();
46         compare_flush_hi();
47         Testing.finishTests();
48     }
49
50     //# Flush1 vs Flush2 (Flush 1 wins highcard is greater

```



```

50 )
51     public static void compare_flushes1() {
52         ArrayList<Card> hand1array = new ArrayList<Card>(
53             Arrays.asList(new Card(13, "S"), new Card(12, "S"), new
54                 Card(9, "S"), new Card(7, "S"), new Card(3, "S")));
55         PokerHand hand1 = new PokerHand(hand1array);
56         PokerHand hand2 = new PokerHand(new ArrayList<
57             Card>(Arrays.asList(new Card(4, "C"), new Card(2, "C"),
58                 new Card(7, "C"), new Card(5, "C"), new Card(10, "C"))));
59
60         int expected_answer = 1;
61         int actual_answer = hand1.compareTo(hand2);
62         Testing.assertEquals("Testing Flush1 vs Flush2 (
63             Flush 1 wins; Highcard is greater)",
64             expected_answer,
65             actual_answer);
66     }
67
68     /// Flush1 vs Flush2 (Flush 2 wins highcard is
69         greater)
70     public static void compare_flushes2() {
71         PokerHand hand1 = new PokerHand(new ArrayList<
72             Card>(Arrays.asList(new Card(4, "C"), new Card(2, "C"),
73                 new Card(7, "C"), new Card(5, "C"), new Card(10, "C"))));
74         PokerHand hand2 = new PokerHand(new ArrayList<
75             Card>(Arrays.asList(new Card(13, "S"), new Card(12, "S"
76                 ), new Card(9, "S"), new Card(7, "S"), new Card(3, "S"
77                 ))));
78
79         int expected_answer = -1;
80         int actual_answer = hand1.compareTo(hand2);
81         Testing.assertEquals("Testing Flush1 vs Flush2 (
82             Flush 2 wins highcard is greater)",
83             expected_answer,
84             actual_answer);
85     }
86
87     /// Flush1 vs Flush2 (Tie)
88     public static void compare_flushes_tie() {
89         PokerHand hand1 = new PokerHand(new ArrayList<
90             Card>(Arrays.asList(new Card(4, "C"), new Card(2, "C"),
91                 new Card(7, "C"), new Card(5, "C"), new Card(10, "C"))));
92         PokerHand hand2 = new PokerHand(new ArrayList<
93             Card>(Arrays.asList(new Card(7, "S"), new Card(5, "S"),
94                 new Card(4, "S"), new Card(2, "S"), new Card(10, "S"))));
95
96         int expected_answer = 0;
97         int actual_answer = hand1.compareTo(hand2);
98         Testing.assertEquals("Testing Flush1 vs Flush2
99             Tie",
100             expected_answer,
101             actual_answer);

```

```

85     }
86
87     /// Flush vs 2 pair
88     public static void compare_flush_2pair() {
89         PokerHand hand1 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(7, "S"), new Card(5, "S"),
new Card(11, "S"), new Card(2, "S"), new Card(10, "S"
)))));
90         PokerHand hand2 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"),
new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));
91
92         int expected_answer = 1;
93         int actual_answer = hand1.compareTo(hand2);
94         Testing.assertEquals("Testing Flush1 vs 2pair (
Flush 1 wins)",
95             expected_answer,
96             actual_answer);
97     }
98
99     /// Flush vs pair
100    public static void compare_flush_pair() {
101        PokerHand hand1 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(7, "S"), new Card(5, "S"),
new Card(4, "S"), new Card(2, "S"), new Card(10, "S"))));
102        PokerHand hand2 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"),
new Card(9, "S"), new Card(10, "C"), new Card(6, "D"))));
103
104        int expected_answer = 1;
105        int actual_answer = hand1.compareTo(hand2);
106        Testing.assertEquals("Testing Flush1 vs pair (
Flush 1 wins)",
107            expected_answer,
108            actual_answer);
109    }
110
111    /// Flush vs high Card
112    public static void compare_flush_hi() {
113        PokerHand hand1 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(7, "S"), new Card(5, "S"),
new Card(4, "S"), new Card(2, "S"), new Card(10, "S"))));
114        PokerHand hand2 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(3, "H"), new Card(4, "D"),
new Card(10, "S"), new Card(8, "C"), new Card(6, "D"))));
115
116        int expected_answer = 1;
117        int actual_answer = hand1.compareTo(hand2);
118        Testing.assertEquals("Testing Flush1 vs highcard
(Flush 1 wins)",
119            expected_answer,
120            actual_answer);

```

```

121     }
122
123     #####2 PAIR TESTS#####
124
125     public static void test_all_two_pair() {
126         Testing.startTests();
127         compare_2pair_flush();
128         compare_2pair_2pair_1();
129         compare_2pair_2pair_2();
130         compare_2pair_2pair_3();
131         compare_2pair_2pair_4();
132         Testing.finishTests();
133     }
134
135     /// 2pair vs Flush
136     public static void compare_2pair_flush() {
137         PokerHand hand1 = new PokerHand(new ArrayList<
138 Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"),
new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));
139
140         PokerHand hand2 = new PokerHand(new ArrayList<
141 Card>(Arrays.asList(new Card(7, "S"), new Card(5, "S"),
new Card(11, "S"), new Card(2, "S"), new Card(10, "S"
142 ))));
143
144         int expected_answer = -1;
145         int actual_answer = hand1.compareTo(hand2);
146         Testing.assertEquals("Testing 2pair vs Flush",
147             expected_answer,
148             actual_answer);
149     }
150
151     /// 2pair1 vs 2pair2 (2pair1 wins higher of pair
152 values is greater)
153     public static void compare_2pair_2pair_1() {
154         PokerHand hand1 = new PokerHand(new ArrayList<
155 Card>(Arrays.asList(new Card(4, "H"), new Card(6, "D"),
new Card(10, "S"), new Card(10, "C"), new Card(4, "D"
156 ))));
157
158         PokerHand hand2 = new PokerHand(new ArrayList<
159 Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"),
new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));
160
161         int expected_answer = 1;
162         int actual_answer = hand1.compareTo(hand2);
163         Testing.assertEquals("Testing 2pair1 vs 2pair2 (
164 2pair1 wins higher of pair values is greater)",
165             expected_answer,
166             actual_answer);
167     }
168
169     /// 2pair1 vs 2pair2 (2pair2 wins higher of pair
170 values is greater)

```

```

160     public static void compare_2pair_2pair_2() {
161         PokerHand hand1 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"),
new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));
162         PokerHand hand2 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(14, "H"), new Card(14, "D"
), new Card(8, "S"), new Card(8, "C"), new Card(6, "D"
))));
163
164         int expected_answer = -1;
165         int actual_answer = hand1.compareTo(hand2);
166         Testing.assertEquals("Testing 2pair1 vs 2pair2 (
2pair2 wins higher of pair values is greater)",
167             expected_answer,
168             actual_answer);
169     }
170
171     ///# 2pair1 vs 2pair2 (2pair1 wins lower of pair
values is greater)
172     public static void compare_2pair_2pair_3() {
173         PokerHand hand1 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"),
new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));
174         PokerHand hand2 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(3, "H"), new Card(4, "D"),
new Card(8, "S"), new Card(8, "C"), new Card(3, "D"))));
175
176         int expected_answer = 1;
177         int actual_answer = hand1.compareTo(hand2);
178         Testing.assertEquals("2pair1 vs 2pair2 (2pair1
wins lower of pair values is greater)",
179             expected_answer,
180             actual_answer);
181     }
182
183     ///# 2pair1 vs 2pair2 (2pair2 wins lower of pair
values is greater)
184     public static void compare_2pair_2pair_4() {
185         PokerHand hand1 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(3, "H"), new Card(4, "D"),
new Card(8, "S"), new Card(8, "C"), new Card(3, "D"))));
186         PokerHand hand2 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"),
new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));
187
188         int expected_answer = -1;
189         int actual_answer = hand1.compareTo(hand2);
190         Testing.assertEquals("Testing 2pair1 vs 2pair2 (
2pair2 wins lower of pair values is greater)",
191             expected_answer,
192             actual_answer);
193     }

```

```

194
195 #####PAIR TESTS#####
196
197     public static void test_all_pair() {
198         Testing.startTests();
199         compare_pair_pair_1();
200         compare_pair_pair_2();
201         compare_pair_pair_3();
202         compare_pair_pair_4();
203         Testing.finishTests();
204     }
205
206
207     /// pair1 vs pair2 (pair1 wins; high pair)
208     public static void compare_pair_pair_1() {
209         PokerHand hand1 = new PokerHand(new ArrayList<
210 Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"),
new Card(9, "S"), new Card(10, "C"), new Card(6, "D"))));
211         PokerHand hand2 = new PokerHand(new ArrayList<
212 Card>(Arrays.asList(new Card(2, "H"), new Card(4, "D"),
new Card(9, "S"), new Card(9, "C"), new Card(6, "D"))));
213
214         int expected_answer = 1;
215         int actual_answer = hand1.compareTo(hand2);
216         Testing.assertEquals("pair1 vs pair2 (pair1 wins
; high pair)",
217             expected_answer,
218             actual_answer);
219     }
220
221     /// pair1 vs pair2 (pair2 wins; high pair)
222     public static void compare_pair_pair_2() {
223         PokerHand hand1 = new PokerHand(new ArrayList<
224 Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"),
new Card(9, "S"), new Card(10, "C"), new Card(6, "D"))));
225         PokerHand hand2 = new PokerHand(new ArrayList<
226 Card>(Arrays.asList(new Card(12, "H"), new Card(4, "D"),
new Card(9, "S"), new Card(12, "C"), new Card(6, "D"))));
227
228         int expected_answer = -1;
229         int actual_answer = hand1.compareTo(hand2);
230         Testing.assertEquals("Testing pair1 vs pair2 (
pair2 wins; high pair)",
231             expected_answer,
232             actual_answer);
233     }
234
235     /// pair1 vs pair2 (pair1 wins; highcard)
236     public static void compare_pair_pair_3() {
237         PokerHand hand1 = new PokerHand(new ArrayList<
238 Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"),
new Card(12, "S"), new Card(10, "C"), new Card(6, "D")
```

```

233 ))));
234         PokerHand hand2 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"),
new Card(9, "S"), new Card(10, "C"), new Card(6, "D"))));
235
236         int expected_answer = 1;
237         int actual_answer = hand1.compareTo(hand2);
238         Testing.assertEquals("Testing pair1 vs pair2 (
pair1 wins; highcard)",
239             expected_answer,
240             actual_answer);
241     }
242
243     /// pair1 vs pair2 (pair2 wins; highcard)
244     public static void compare_pair_pair_4() {
245         PokerHand hand1 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"),
new Card(9, "S"), new Card(10, "C"), new Card(6, "D"))));
246         PokerHand hand2 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(10, "H"), new Card(13, "D"
), new Card(9, "S"), new Card(10, "C"), new Card(6, "D"
))));
247
248         int expected_answer = -1;
249         int actual_answer = hand1.compareTo(hand2);
250         Testing.assertEquals("Testing pair1 vs pair2 (
pair2 wins; highcard",
251             expected_answer,
252             actual_answer);
253     }
254 }
255
256
257
258
259
260 /*
261
262
263
264 (_____) ( ) ( ) (_____) (_____) ( ) ( ) (_____)
265 ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( )
266 ( ) (_____) ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( )
267 ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( ) ( )
268 ( ) ( ) ( ) (_____) (_____) ( ) ( ) (_____)
269
270
271 */
272
273
274
275

```