```java
package proj3; // do not erase. Gradescope expects this.

/**
 * Class to model a single playing card
 *
 * @author Neil Datero
 */
public class Card {

    private int rank;
    private String suit;
    private final int JACK = 11;
    private final int QUEEN = 12;
    private final int KING = 13;
    private final int ACE = 14;
    public static final String[] SUITS = {"Hearts", "Spades", "Diamonds", "Clubs"};
    public static final int[] RANKS = {2,3,4,5,6,7,8,9,10,11,12,13,14};

    /**
     * Non-default constructor for playing card
     * @param rankOfCard Rank of Card as an integer from 2-14
     * @param suitOfCard Suit of Card as a full string. i.e "Spades", "Hearts" etc.
     */
    public Card(int rankOfCard, String suitOfCard) {
        rank = rankOfCard;
        suit = suitOfCard;
    }
    /**
     * Getter function for rank of card
     * @return Rank as an integer from 2-14
     */
    public int getRank() {
        return rank;
    }

```

```
36      /**
37       * Getter function for string of card
38       * @return Suit as a string, "Hearts", "Spades
    ", etc.
39       */
40      public String getSuit(){
41          return suit;
42      }
43
44
45      /**
46       * Translates the rank into a string. If the
    rank is greater than 10, it will return the string
    version of the face card.
47       * @return Rank in a string form: 2-10 or "
    Hearts", "Spades" etc.
48       */
49      private String rankAsString() {
50          int cardRank = getRank();
51          String rankAsString;
52
53          if (cardRank == JACK) { rankAsString = "
    Jack"; }
54          else if (cardRank == QUEEN) { rankAsString
     = "Queen"; }
55          else if (cardRank == KING) { rankAsString
     = "King"; }
56          else if (cardRank == ACE) { rankAsString =
    "Ace"; }
57          else { rankAsString = String.valueOf(
    cardRank); }
58
59          return rankAsString;
60      }
61
62
63
64      /**
65       * Returns a string representation of the
    object.
66       * @return Returns a string representation of
```

```
66 the object.
67      */
68     public String toString(){
69         return rankAsString() + " of " + getSuit
   ();
70     }
71
72
73
74 }
75
```

```java
package proj3; // do not erase. Gradescope expects this.

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


/**
 * Class that models a deck of playing cards
 *
 * @author Neil Daterao
 */
public class Deck {
    private ArrayList<Card> deckOfCards = new ArrayList<Card>();
    private int nextToDeal;
    public static final int MAXCARDAMOUNT = 52;


    /**
     * Default Constructor. Intitializes a standard playing card deck of 52 cards.
     */
    public Deck() {
        nextToDeal = 0;
        for (String suit : Card.SUITS) {
            for (int rank : Card.RANKS) {
                deckOfCards.add(new Card(rank,suit));
            }
        }
    }

    /**
     * Method to shuffle the current deck.
     */
    public void shuffle() {
        if (nextToDeal < MAXCARDAMOUNT) {
            for (int currentCardIndex = nextToDeal; currentCardIndex < deckOfCards.size(); currentCardIndex++)  {
```

```
36                  int randomCardIndex =
   ThreadLocalRandom.current().nextInt(nextToDeal,
   deckOfCards.size());
37                  Card currentCard = deckOfCards.get(
   currentCardIndex);
38                  Card randomCardToSwap = deckOfCards
   .get(randomCardIndex);
39                  deckOfCards.set(currentCardIndex,
   randomCardToSwap);
40                  deckOfCards.set(randomCardIndex,
   currentCard);
41
42              }
43          }
44          else { System.out.println("No cards left in
    the deck to shuffle!");}
45      }
46
47      /**
48       *
49       * @return Returns the next undealt card. If
   all cards have been dealt, will return null.
50       * @apiNote This runs in O(1) time since
   ArrayLists have constant look up times and we're
   not shifting the index of any cards in the
   ArrayList.
51       */
52      public Card deal(){
53          if (nextToDeal < MAXCARDAMOUNT) {
54              Card cardToBeDealt = deckOfCards.get(
   nextToDeal);
55              nextToDeal++;
56              return cardToBeDealt;
57          }
58
59          else { return null; }
60
61      }
62
63      /**
64       *
```

```
65      * @return Returns true if deck is empty and
   false if there are still undealt cards.
66      */
67     public boolean isEmpty() {
68         return nextToDeal == MAXCARDAMOUNT;
69     }
70 
71     /**
72      *
73      * @return An integer representing the amount
   of undealt cards in the deck
74      */
75     public int size() {
76         return MAXCARDAMOUNT - nextToDeal;
77     }
78 
79     /**
80      *
81      * Resets the deck to a state where all cards
   are undealt.
82      */
83     public void gather() {
84         nextToDeal = 0;
85     }
86 
87 
88     /**
89      * @return Returns the string version of all
   the undealt cards in the deck with one playing
   card per line
90      */
91     public String toString() {
92         if (nextToDeal < MAXCARDAMOUNT) {
93             String deckOfCardsAsString = "";
94 
95             for (int indexOfCard = nextToDeal;
   indexOfCard < deckOfCards.size(); indexOfCard
   ++) {
96                 deckOfCardsAsString += deckOfCards
   .get(indexOfCard).toString() + "\n";
97             }
```

```
 98
 99              return deckOfCardsAsString;
100          }
101          else { return "Deck is empty!"; }
102      }
103
104
105
106
107 }
108
109
```

```java
package proj3;


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

/**
 * Simple game where the user chooses which poker hand they think is stronger.
 *
 * @author Neil Daterao
 */

public class Client {
    public static void main(String[] args) {
        String playAgain = "Y";

        while (playAgain.equals("Y")) {
            Scanner inputChecker = new Scanner(System.in);

            System.out.println("Welcome to the poker strength test game! Press return to get your first set of hands! \n");
            inputChecker.nextLine();

            Deck deckOfCards = new Deck();
            deckOfCards.shuffle();
            int totalPoints = 0;
            boolean gameIsOver = false;

            while (deckOfCards.size() >= PokerHand.MAXCARDSINPOKERHAND * 2 && gameIsOver == false) {
                ArrayList<Card> pokerHand = new ArrayList<Card>();
                ArrayList<Card> otherPokerHand = new ArrayList<Card>();

                for (int i = 0 ; i < PokerHand.MAXCARDSINPOKERHAND; i++) {
                    pokerHand.add(deckOfCards.deal
```

```
33 ());
34                     otherPokerHand.add(deckOfCards.
   deal());
35                 }
36 
37                 PokerHand pokerHandObj = new
   PokerHand(pokerHand);
38                 PokerHand otherPokerHandObj = new
   PokerHand(otherPokerHand);
39 
40                 int expectedWinner = pokerHandObj.
   compareTo(otherPokerHandObj);
41 
42                 System.out.println("    Hand #1 \n"
   );
43                 System.out.println(
   "---------------");
44                 System.out.println(pokerHandObj);
45                 System.out.println(
   "---------------");
46                 System.out.println("    Hand #2 \n"
   );
47                 System.out.println(
   otherPokerHandObj);
48                 System.out.println(
   "---------------");
49                 System.out.println("Cheat:" +
   expectedWinner); //-> To check while testing
50 
51                 System.out.println("Which hand do
   you think is stronger, 1 or 2? If you think it's a
   tie enter 0!");
52                 String answer = inputChecker.
   nextLine();
53 
54                 String[] acceptedAnswers = {"0", "1
   ", "2"};
55 
56                 while (!isValidAnswer(answer,
   acceptedAnswers)) {
57                     System.out.println("Invalid
```

```
57 Input!");
58                     System.out.println("Which hand
   do you think is stronger, 1 or 2? If you think it's
    a tie enter 0!");
59                     answer = inputChecker.nextLine
   ();
60                 }
61 
62                 int chosenAnswer = Integer.parseInt
   (answer);
63                 if (chosenAnswer == 1 &&
   expectedWinner > 0) {
64                     totalPoints++;
65                     System.out.println("\nGood job
   you got it right! Let's see if you can get the next
    set of hands right too!\n");
66                 }
67                 else if (chosenAnswer == 2 &&
   expectedWinner < 0) {
68                     totalPoints++;
69                     System.out.println("\nGood job
   you got it right! Let's see if you can get the next
    set of hands right too!\n");
70                 }
71                 else if (chosenAnswer == 0 &&
   expectedWinner == 0) {
72                     totalPoints++;
73                     System.out.println("\nGood job
   you got it right! Let's see if you can get the next
    set of hands right too!\n");
74                 }
75                 else {
76                     System.out.println("Incorrect!"
   );
77                     gameIsOver = true;
78                 }
79             }
80 
81             System.out.println("Game Over!");
82             System.out.println("Total Points: " +
   totalPoints);
```

```
 83
 84             System.out.println("Would you like to
    play again (Y/N)?");
 85             playAgain = inputChecker.nextLine();
 86
 87             if (!playAgain.equals("Y")){
 88                 System.out.println("Awh Bummer!
    Have a nice day!");
 89                 inputChecker.close();
 90                 return;
 91             }
 92         }
 93
 94     }
 95
 96
 97     private static boolean isValidAnswer(String
    answer, String[] acceptedAnswers) {
 98         for (String acceptedAnswer :
    acceptedAnswers) {
 99             if (answer.equals(acceptedAnswer)) {
100                 return true;
101             }
102         }
103         return false;
104     }
105 }
106
```

```
 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  * <code>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 printed.
157      */
158     public static void testSection(String
    sectionTitle)
159     {
160         if (VERBOSE) {
161             int dashCount = sectionTitle.length();
162             System.out.println(sectionTitle);
163             for (int i = 0; i < dashCount; i++) {
164                 System.out.print("-");
165             }
166             System.out.println();
167             System.out.println();
168         }
169     }
170 
171     /**
172      * Initializes the test suite. Should be
    called before running any
173      * tests, so that passes and fails are
```

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

```java
package proj3; // do not erase. Gradescope expects this.

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.Map;
import java.util.Arrays;


/**
 * Class that models a standard poker hand
 * @author Neil Daterao
 */
public class PokerHand {

    public static final int MAXCARDSINPOKERHAND = 5;
    private ArrayList<Card> pokerHandContents = new ArrayList<Card>();
    private final int FIRSTVALIDCARDINDEX = 0;
    private final int FLUSH = 4;
    private final int TWOPAIR = 3;
    private final int ONEPAIR = 2;
    private final int HIGHCARD = 1;
    private final int DEFAULTCARDRANK = 2;


    /**
     * Non-default constructor. Intializes a poker hand of cards passed in as a parameter
     * @param cardList An ArrayList of cards objects
     */
    public PokerHand(ArrayList<Card> cardList) {
        for (Card card : cardList) { pokerHandContents.add(card);  }
    }

    /**
     * Nothing will happen if poker hand already
```

```
35 has 5 cards
36      * @param cardToBeAdded Card object that you
   want to add to the poker hand
37      */
38     public void addCard(Card cardToBeAdded){
39         if (size() < MAXCARDSINPOKERHAND) {
   pokerHandContents.add(cardToBeAdded); }
40     }
41 
42     /**
43      *
44      * @return Integer representing the size of the
    poker hand
45      */
46     public int size() {
47         return pokerHandContents.size();
48     }
49 
50 
51     /**
52      *
53      * @param index Integer representing index of
   card. Index must be >= 0 and less than 5, the max
   cards in the hand.
54      * @return Card object at given index
55      */
56     public Card get_ith_card(int index) {
57         if (index >= FIRSTVALIDCARDINDEX && index
    < MAXCARDSINPOKERHAND) {
58             return pokerHandContents.get(index);
59         }
60         else { return null; }
61     }
62 
63     /**
64      * @return String version of the poker hand
   with one card per line
65      */
66     public String toString() {
67         String stringVersionOfPokerHand = "";
68         for (int i = FIRSTVALIDCARDINDEX; i < size
```

```
68 (); i++) {
69             stringVersionOfPokerHand +=
   get_ith_card(i).toString() + "\n";
70         }
71         return stringVersionOfPokerHand;
72     }
73 
74    /**
75     * A flush is if 5 cards in a poker hand have
   the same suit. Thus, if 4 cards have the same suit
   , the hand is not classified as a flush because a
   flush requires 5 cards.
76     * @return True if poker hand is a flush and
   false if not.
77     */
78    private boolean isFlush(){
79         Card firstCard = get_ith_card(
   FIRSTVALIDCARDINDEX);
80         String previousSuit = firstCard.getSuit
   ();
81         for (Card card : pokerHandContents) {
82             if (!card.getSuit().equals(
   previousSuit)) {
83                 return false;
84             }
85         }
86         if (size() == MAXCARDSINPOKERHAND) {
   return true; }
87         else { return false; }
88    }
89 
90    /**
91     *
92     * @return Integer returning the number of
   pairs in the poker hand
93     */
94    private int numberOfPairs() {
95 
96         int[] rankOfCards = getRanksOfHandInArray
   ();
97         int pairs = 0;
```

```
 98
 99
100         Map<Integer, Integer> countOfRanks =
    ArrayUtilities.Counter(rankOfCards);
101
102         for (Integer rank : countOfRanks.keySet
    ()) {
103             if (countOfRanks.get(rank).equals(4
    )) { pairs += 2; }
104             else if (countOfRanks.get(rank) >= 2
    ) { pairs += 1; }
105             else { pairs += 0; }
106
107         }
108
109         return pairs;
110     }
111
112     /**
113      *
114      * @return Classification of hand as power
    ranking. 4: Flush 3: Two Pair 2: One pair 1: High
    Card
115      */
116     public int determineClassificationOfHand() {
117         if (isFlush()){ return FLUSH; }
118         else if (numberOfPairs() == 2) { return
    TWOPAIR; }
119         else if (numberOfPairs() == 1) { return
    ONEPAIR; }
120         else { return HIGHCARD; }
121
122     }
123
124     /**
125      *
126      * @return Ranks of Hand in an integer array.
127      */
128     private int[] getRanksOfHandInArray() {
129         int[] ranksOfHandInArray = new int[size()];
130
```

```
131         for (int i = FIRSTVALIDCARDINDEX; i < size
    (); i++) {
132             ranksOfHandInArray[i] = get_ith_card(i
    ).getRank();
133         }
134         return ranksOfHandInArray;
135 
136     }
137     /**
138      * Function to determine which hand of two
    pairs is stronger
139      * @param otherHand Another PokerHand Object
    which is classified as a two pair
140      * @return Will return 1 if this PokerHand is
    stronger, -1 if the otherHand PokerHand is
    stronger and 0 if they are of equal strength
141      */
142     private int determineWinningTwoPair(PokerHand
    otherHand) {
143         ArrayList<Integer> pairsInThisHand = new
    ArrayList<Integer>();
144         int extraCardRankInThisHand =
    DEFAULTCARDRANK;
145         ArrayList<Integer> pairsInOtherHand = new
    ArrayList<Integer>();
146         int extraCardRankInOtherHand =
    DEFAULTCARDRANK;
147         int maxIndex = 2; //Magic number but since
    the function is private it's okay! The reason for
    this magic number is to account for hands where we
     are comparing 4 of a kind to a 2 regular 2 pair.
148 
149         int[] ranksInThisHand =
    getRanksOfHandInArray();
150         int[] ranksInOtherHand = otherHand.
    getRanksOfHandInArray();
151 
152         Map<Integer, Integer>
    countsOfRanksInThisHand = ArrayUtilities.Counter(
    ranksInThisHand);
153         Map<Integer, Integer>
```

```
153 countsOfRanksInOtherHand = ArrayUtilities.Counter(
    ranksInOtherHand);
154 
155         for (int rank : countsOfRanksInThisHand.
    keySet()) {
156             if (countsOfRanksInThisHand.get(rank
    ) >= 2) {
157                 pairsInThisHand.add(rank);
158             }
159             else { extraCardRankInThisHand = rank
    ; }
160         }
161 
162         for (int rank : countsOfRanksInOtherHand.
    keySet()) {
163          if (countsOfRanksInOtherHand.get(rank) >=
    2) {
164              pairsInOtherHand.add(rank);
165          }
166          else { extraCardRankInOtherHand = rank; }
167      }
168          Collections.sort(pairsInThisHand,
    Comparator.reverseOrder());
169          Collections.sort(pairsInOtherHand,
    Comparator.reverseOrder());
170 
171          if (pairsInThisHand.size() !=
    pairsInOtherHand.size()) { maxIndex = 1; }
172 
173          for (int i=FIRSTVALIDCARDINDEX; i <
    maxIndex; i++) {
174              if (pairsInThisHand.get(i) >
    pairsInOtherHand.get(i)) { return 1; }
175              else if (pairsInThisHand.get(i) <
    pairsInOtherHand.get(i)) { return -1; }
176          }
177          return extraCardRankInThisHand -
    extraCardRankInOtherHand;
178     }
179 
180     /**
```

```
181     * Function to determine which hand of one
    pairs is stronger
182     * @param otherHand Another PokerHand Object
    which is classified as One Pair
183     * @return Will return 1 if this PokerHand is
    stronger, -1 if the otherHand PokerHand is
    stronger and 0 if they are of equal strength
184     */
185    private int determineWinningOnePair(PokerHand
    otherHand) {
186         int pairInThisHand = DEFAULTCARDRANK;
187         ArrayList<Integer> extraCardsInThisHand =
    new ArrayList<Integer>();
188         int pairInOtherHand = DEFAULTCARDRANK;
189         ArrayList<Integer> extraCardsInOtherHand
     = new ArrayList<Integer>();
190         int maxIndex = 3; //Magic number again,
    but it's okay since it is a private function, this
     is to account for comparing three of a kind with
    one pair, the list of extra cards will be
    different lengths and we want to iterate through
    the smallest sorted array.
191
192         int[] ranksInThisHand =
    getRanksOfHandInArray();
193         int[] ranksInOtherHand = otherHand.
    getRanksOfHandInArray();
194
195         Map<Integer, Integer>
    countsOfRanksInThisHand = ArrayUtilities.Counter(
    ranksInThisHand);
196         Map<Integer, Integer>
    countsOfRanksInOtherHand = ArrayUtilities.Counter(
    ranksInOtherHand);
197
198         for (int rank : countsOfRanksInThisHand.
    keySet()) {
199             if (countsOfRanksInThisHand.get(rank
    ) >= 2 ) {
200                 pairInThisHand = rank;
201             }
```

```
202             else { extraCardsInThisHand.add(rank
    ); }
203         }
204 
205         for (int rank : countsOfRanksInOtherHand.
    keySet()) {
206             if (countsOfRanksInOtherHand.get(rank
    ) >= 2 ) {
207                 pairInOtherHand = rank;
208             }
209             else { extraCardsInOtherHand.add(rank
    ); }
210         }
211 
212         Collections.sort(extraCardsInThisHand,
    Comparator.reverseOrder());
213         Collections.sort(extraCardsInOtherHand,
    Comparator.reverseOrder());
214 
215         if (extraCardsInThisHand.size() !=
    extraCardsInOtherHand.size()) { maxIndex = 2; }
216 
217         if (pairInThisHand == pairInOtherHand) {
218             for (int i = FIRSTVALIDCARDINDEX; i <
    maxIndex; i++) {
219                 if (extraCardsInThisHand.get(i) >
    extraCardsInOtherHand.get(i)) { return 1; }
220                 else if (extraCardsInThisHand.get(
    i) < extraCardsInOtherHand.get(i)) { return -1; }
221             }
222         }
223 
224         return pairInThisHand - pairInOtherHand;
225     }
226 
227  /**
228     * Function to determine which hand of High
    Cards is stronger
229     * @param otherHand Another PokerHand Object
    which is classified as High Card
230     * @return Will return 1 if this PokerHand is
```

```
230 stronger, -1 if the otherHand PokerHand is
    stronger and 0 if they are of equal strength
231     */
232    private int determineWinningHighCard(PokerHand
    otherHand) {
233         int[] ranksInThisHand =
    getRanksOfHandInArray();
234         int[] ranksInOtherHand = otherHand.
    getRanksOfHandInArray();
235 
236         Arrays.sort(ranksInThisHand);
237         Arrays.sort(ranksInOtherHand);
238 
239         for (int i = size() - 1; i >= 0 ; i--) {
240             if (ranksInThisHand[i] >
    ranksInOtherHand[i]) { return 1; }
241             else if (ranksInThisHand[i] <
    ranksInOtherHand[i]) { return -1; }
242 
243         }
244         return 0;
245 
246    }
247 
248    /**
249     * Function to determine which hand of Flushes
    is stronger
250     * @param otherHand Another PokerHand Object
    which is classified as a Flush
251     * @return Will return 1 if this PokerHand is
    stronger, -1 if the otherHand PokerHand is
    stronger and 0 if they are of equal strength
252     */
253    private int determineWinningFlush(PokerHand
    otherHand) {
254        //Since we alreay know this is a flush, we
    have to check the ranks of the card and we can use
     the same algorithm we used to check high cards.
255         return determineWinningHighCard(otherHand
    );
256    }
```

```
257
258
259
260     /**
261     * Determines how this hand compares to another
     hand, returns
262     * positive, negative, or zero depending on the
     comparison.
263     *
264     * @param other The hand to compare this hand
    to
265     * @return a negative number if this is worth
    LESS than other, zero
266     * if they are worth the SAME, and a positive
    number if this is worth
267     * MORE than other
268     */
269     public int compareTo(PokerHand other) {
270
271         if (determineClassificationOfHand() -
    other.determineClassificationOfHand() == 0) {
272             if (determineClassificationOfHand
    () == FLUSH) { return determineWinningFlush(other
    ); }
273             else if (determineClassificationOfHand
    () == TWOPAIR) { return determineWinningTwoPair(
    other); }
274             else if (determineClassificationOfHand
    () == ONEPAIR) { return determineWinningOnePair(
    other); }
275             else if (determineClassificationOfHand
    () == HIGHCARD) { return determineWinningHighCard(
    other); }
276
277         }
278         return determineClassificationOfHand() -
    other.determineClassificationOfHand();
279     }
280
281
282
```

```
283
284
285 }
286
```

```java
package proj3;

/**
 * Class which performs unit testing on the Card class
 *
 * @author Neil Daterao
 */
public class CardTester {

    /**
     * Foreman function which calls all tests.
     *
     */
    public static void main(String[] args) {
        Testing.setVerbose(true);
        Testing.startTests();
        Testing.testSection("Testing getRank");
        CardTester.testGetRank();
        Testing.testSection("Testing getSuit");
        CardTester.testGetSuit();
        Testing.testSection("Testing toString");
        CardTester.testToString();
        Testing.finishTests();
    }

    /**
     * Tests for getRank
     */
    private static void testGetRank(){
        Card cardWithRank3 = new Card(3, "Hearts");
        int expected = 3;
        int actual = cardWithRank3.getRank();
        Testing.assertEquals("Testing get rank with a card with rank 3", expected, actual);


        Card cardWithRank6 = new Card(6, "Hearts");
        expected = 6;
        actual = cardWithRank6.getRank();
        Testing.assertEquals("Testing get rank with
```

```
39  a card with rank 6", expected, actual);
40
41          Card cardWithRank14 = new Card(14, "
   Diamonds");
42          expected = 14;
43          actual = cardWithRank14.getRank();
44          Testing.assertEquals("Testing get rank with
    a card with rank 14", expected, actual);
45
46
47      }
48
49      /**
50       * Test for getSuit
51       */
52      private static void testGetSuit() {
53          Card cardWithSuitDiamonds = new Card(4, "
   Diamonds");
54          String expected = "Diamonds";
55          String actual = cardWithSuitDiamonds.
   getSuit();
56          Testing.assertEquals("Testing get suit with
    a card with suit Diamonds", expected, actual);
57
58          Card cardWithSuitHearts = new Card(4, "
   Hearts");
59          String expectedHearts = "Hearts";
60          String actualHearts = cardWithSuitHearts.
   getSuit();
61          Testing.assertEquals("Testing get suit with
    a card with suit Hearts", expectedHearts,
   actualHearts);
62
63          Card cardWithSuitSpades = new Card(7, "
   Spades");
64          String expectedSpades = "Spades";
65          String actualSpades = cardWithSuitSpades.
   getSuit();
66          Testing.assertEquals("Testing get suit with
    a card with suit Spades", expectedSpades,
   actualSpades);
```

```
67
68          Card cardWithSuitClubs = new Card(9, "
    Clubs");
69          String expectedClubs = "Clubs";
70          String actualClubs = cardWithSuitClubs.
    getSuit();
71          Testing.assertEquals("Testing get suit
    with a card with suit Clubs", expectedClubs,
    actualClubs);
72
73      }
74
75      /**
76       * Tests for toString
77       */
78      private static void testToString() {
79          Card card3Clubs = new Card(3, "Clubs");
80          String expected = "3 of Clubs";
81          String actual = card3Clubs.toString();
82          Testing.assertEquals("Testing card of 3 of
     clubs", expected, actual);
83
84          Card cardAceSpades = new Card(14, "Spades"
    );
85          String expectedAceSpades = "Ace of Spades"
    ;
86          String actualAceSpades = cardAceSpades.
    toString();
87          Testing.assertEquals("Testing card of Ace
    of Spades", expectedAceSpades, actualAceSpades);
88
89          Card cardKingDiamonds = new Card(13, "
    Diamonds");
90          String expectedKingDiamonds = "King of
    Diamonds";
91          String actualKingDiamonds =
    cardKingDiamonds.toString();
92          Testing.assertEquals("Testing card of King
     of Diamonds", expectedKingDiamonds,
    actualKingDiamonds);
93
```

```
 94          Card cardQueenHearts = new Card(12, "
    Hearts");
 95          String expectedQueenHearts = "Queen of
    Hearts";
 96          String actualQueenHearts = cardQueenHearts
    .toString();
 97          Testing.assertEquals("Testing card of
    Queen of Hearts", expectedQueenHearts,
    actualQueenHearts);
 98
 99          Card cardJackClubs = new Card(11, "Clubs"
    );
100          String expectedJackClubs = "Jack of Clubs"
    ;
101          String actualJackClubs = cardJackClubs.
    toString();
102          Testing.assertEquals("Testing card of Jack
     of Clubs", expectedJackClubs, actualJackClubs);
103
104          Card card10Diamonds = new Card(10, "
    Diamonds");
105          String expected10Diamonds = "10 of
    Diamonds";
106          String actual10Diamonds = card10Diamonds.
    toString();
107          Testing.assertEquals("Testing card of 10
    of Diamonds", expected10Diamonds, actual10Diamonds
    );
108
109          Card card6Spades = new Card(6, "Spades");
110          String expected6Spades = "6 of Spades";
111          String actual6Spades = card6Spades.
    toString();
112          Testing.assertEquals("Testing card of 6 of
     Spades", expected6Spades, actual6Spades);
113      }
114 }
115
```

```java
package proj3;

/**
 * Class which performs unit testing on the Deck class
 *
 * @author Neil Daterao
 */
public class DeckTester {
    /**
     *
     * Foreman function which runs all the tests
     */
    public static void main(String[] args) {
        Testing.setVerbose(true);
        Testing.startTests();
        Testing.testSection("Testing shuffle method");
        DeckTester.testShuffle();
        Testing.testSection("Testing deal method");
        DeckTester.testDeal();
        Testing.testSection("Testing isEmpty");
        DeckTester.testIsEmpty();
        Testing.testSection("Testing size method");
        DeckTester.testSize();
        Testing.testSection("Testing gather method");
        DeckTester.testGather();
        Testing.finishTests();

    }

    private static void testShuffle() {
        Deck unshuffledDeck = new Deck();
        Deck shuffledDeck = new Deck();
        shuffledDeck.shuffle();

        Testing.assertTrue("Testing shuffle function", unshuffledDeck.toString() != shuffledDeck.toString());
    }
```

```
37
38      private static void testDeal() {
39          Deck newDeck = new Deck();
40          Card cardToBeDealt = newDeck.deal();
41          String expected = "2 of Hearts";  //first
    card of deck
42          Testing.assertEquals("Testing deal",
    expected, cardToBeDealt.toString());
43
44          int i = 0;
45          while (i < Deck.MAXCARDAMOUNT - 1) {
46              cardToBeDealt = newDeck.deal();
47              i++;
48          }
49          expected = "Ace of Clubs"; //last card of
    deck
50          Testing.assertEquals("Testing dealing of
    last card", expected, cardToBeDealt.toString());
51
52          cardToBeDealt = newDeck.deal();
53          expected = null; //no cards left to deal
54          Testing.assertEquals("Testing to ensure
    null gets returned if no cards left", expected,
    cardToBeDealt);
55
56      }
57
58      private static void testIsEmpty(){
59          Deck newDeck = new Deck();
60          boolean actual = newDeck.isEmpty();
61          Testing.assertFalse("Testing a non empty
    deck", actual);
62
63          int i = 0;
64          while (i < Deck.MAXCARDAMOUNT) {
65              newDeck.deal();
66              i++;
67          }
68          actual = newDeck.isEmpty();
69          Testing.assertTrue("Testing an empty deck"
    , actual);
```

```
 70     }
 71
 72     private static void testSize() {
 73         Deck newDeck = new Deck();
 74         int expected = 52;
 75         int actual = newDeck.size();
 76         Testing.assertEquals("Testing size of a
    full deck", expected, actual);
 77
 78         newDeck.deal();
 79         expected = 51;
 80         actual = newDeck.size();
 81         Testing.assertEquals("Testing size of a
    deck without one card", expected, actual);
 82
 83         int i = 0;
 84         while (i < 50) {
 85             newDeck.deal();
 86             i++;
 87         }
 88         expected = 1;
 89         actual = newDeck.size();
 90         Testing.assertEquals("Testing size of a
    deck with one card", expected, actual);
 91
 92         newDeck.deal();
 93         expected = 0;
 94         actual = newDeck.size();
 95         Testing.assertEquals("Testing size of a
    deck with no cards", expected, actual);
 96     }
 97
 98     private static void testGather(){
 99         Deck newDeck = new Deck();
100         int i = 0;
101         while (i < 50) {
102             newDeck.deal();
103             i++;
104         }
105         newDeck.gather();
106         int expected = 52;
```

```
107         int actual = newDeck.size();
108         Testing.assertEquals("Testing gathering
    all the cards after they've been dealt", expected
    , actual);
109     }
110 }
111
```

```java
package proj3;



import java.util.HashMap;
import java.util.Map;

/**
 * Class with helpful Array Utilities, most notably "Counter". This has the same behavior as Counter from collections in Python.
 *
 * @author Neil Daterao
 */
public class ArrayUtilities {
    /**
     * Counter is identical to the Counter from collections in Python. Java doesn't have it's own counter thus I created my own method. It returns a map where the key in the number and the value is the amount of occurances of the number.
     * @param listOfNums A list of integers
     * @return Returns a map where the key in the number and the value is the amount of occurances of the number.
     */
    public static Map<Integer, Integer> Counter(int[] listOfNums){
        final Map<Integer, Integer> numPairedWithCount = new HashMap<>();

        for (int i = 0; i < listOfNums.length; i++) {
            numPairedWithCount.put(listOfNums[i], numPairedWithCount.getOrDefault(listOfNums[i], 0) + 1);
        }

        return numPairedWithCount;
    }

```

```
29 }
30
```

```java
package proj3;

import java.util.ArrayList;
import java.util.Arrays;

/**
 * Class which performs unit testing on the PokerHand class
 *
 * @author Neil Daterao
 * @note I affirm that I have carried out the attached academic endeavors with full academic honesty, in
accordance with the Union College Honor Code and the course syllabus.
 */
public class PokerComparisonTests {

    public static void main(String[] args) {
        Testing.setVerbose(true);
        Testing.startTests();
        Testing.testSection("Testing compareTo method");
        PokerComparisonTests.testCompareTo();
        Testing.finishTests();

    }

    private static void testCompareTo() {
        PokerComparisonTests.testOnePair4vAce();
        PokerComparisonTests.testTwoPair94v93();
        PokerComparisonTests.testOnePairAceHighCardWin();
        PokerComparisonTests.testTwoPair58v76();
        PokerComparisonTests.testTwoPairOfSameDiffHighCard();
        PokerComparisonTests.test9HighFlushv6HighFlush();
        PokerComparisonTests.testKing107FlushvKing105Flush();
        PokerComparisonTests.testFlushActualTie();
```

```
33          PokerComparisonTests.testHighCardTie();
34          PokerComparisonTests.testTwoPairTie();
35          PokerComparisonTests.testOnePairTie();
36          PokerComparisonTests.testFlushVHighCard();
37          PokerComparisonTests.testTwoPairvOnePair();
38          PokerComparisonTests.testFullHousevStraight
    ();
39          PokerComparisonTests.
    testRoyalFlushvStraightFlush();
40          PokerComparisonTests.testFourOfAKindV2Pair
    ();
41          PokerComparisonTests.
    testHighCardTieUpToThirdCard();
42          PokerComparisonTests.
    testOnePairTieTillFourthCard();
43          PokerComparisonTests.
    testThreeOfAKindVsTwoPair();
44
45
46
47      }
48
49      private static void testOnePair4vAce() {
50          ArrayList<Card> pokerHand = new ArrayList<
    Card>(Arrays.asList(
51              new Card(14, "Hearts"),
52              new Card(14, "Spades"),
53              new Card(11, "Diamonds"),
54              new Card(4, "Spades"),
55              new Card(3, "Clubs")));
56
57          ArrayList<Card> otherPokerHand = new
    ArrayList<Card>(Arrays.asList(
58              new Card(4, "Hearts"),
59              new Card(4, "Spades"),
60              new Card(11, "Diamonds"),
61              new Card(2, "Spades"),
62              new Card(3, "Clubs")));
63
64          PokerHand pokerHandObj = new PokerHand(
    pokerHand);
```

```
65          PokerHand otherPokerHandObj = new
    PokerHand(otherPokerHand);
66          int expected = 10;
67          int actual = pokerHandObj.compareTo(
    otherPokerHandObj);
68          Testing.assertEquals("Testing one pair,
    aces v 4s", expected, actual);
69      }
70
71      private static void testTwoPair94v93() {
72          ArrayList<Card> pokerHand2 = new ArrayList
    <Card>(Arrays.asList(
73              new Card(9, "Hearts"),
74              new Card(9, "Spades"),
75              new Card(4, "Diamonds"),
76              new Card(4, "Spades"),
77              new Card(3, "Clubs")
78          ));
79          ArrayList<Card> otherPokerHand2 = new
    ArrayList<Card>(Arrays.asList(
80              new Card(9, "Diamonds"),
81              new Card(9, "Clubs"),
82              new Card(5, "Spades"),
83              new Card(3, "Hearts"),
84              new Card(3, "Spades")
85          ));
86          PokerHand PokerHandObj2 = new PokerHand(
    pokerHand2);
87          PokerHand otherPokerHandObj2 = new
    PokerHand(otherPokerHand2);
88          int expected = 1;
89          int actual = PokerHandObj2.compareTo(
    otherPokerHandObj2);
90          Testing.assertEquals("Testing two pairs,
    9s and 4s vs 9s and 3s", expected, actual);
91      }
92
93      private static void testOnePairAceHighCardWin
    () {
94          ArrayList<Card> pokerHand3 = new ArrayList
    <Card>(Arrays.asList(
```

```
 95              new Card(14, "Hearts"),
 96              new Card(14, "Spades"),
 97              new Card(10, "Diamonds"),
 98              new Card(7, "Spades"),
 99              new Card(2, "Clubs")
100          ));
101          ArrayList<Card> otherPokerHand3 = new
    ArrayList<Card>(Arrays.asList(
102              new Card(14, "Diamonds"),
103              new Card(14, "Clubs"),
104              new Card(11, "Spades"),
105              new Card(8, "Hearts"),
106              new Card(3, "Spades")
107          ));
108          PokerHand pokerHandObj3 = new PokerHand(
    pokerHand3);
109          PokerHand otherPokerHandObj3 = new
    PokerHand(otherPokerHand3);
110          int expected = -1;
111          int actual = pokerHandObj3.compareTo(
    otherPokerHandObj3);
112          Testing.assertEquals("Testing two pairs of
     Aces with high cards determining the winner",
    expected, actual);
113      }
114
115      private static void testTwoPair58v76() {
116          ArrayList<Card> pokerHand = new ArrayList<
    Card>(Arrays.asList(
117              new Card(5, "Hearts"),
118              new Card(5, "Spades"),
119              new Card(8, "Diamonds"),
120              new Card(8, "Spades"),
121              new Card(10, "Clubs")
122          ));
123          ArrayList<Card> otherPokerHand = new
    ArrayList<Card>(Arrays.asList(
124              new Card(7, "Diamonds"),
125              new Card(7, "Clubs"),
126              new Card(6, "Spades"),
127              new Card(6, "Hearts"),
```

```
128             new Card(3, "Diamonds")
129         ));
130         PokerHand pokerHandObj = new PokerHand(
    pokerHand);
131         PokerHand otherPokerHandObj = new
    PokerHand(otherPokerHand);
132         int expected = 1;
133         int actual = pokerHandObj.compareTo(
    otherPokerHandObj);
134         Testing.assertEquals("Testing two pairs of
     5 and 8 vs 7 and 6", expected, actual);
135     }
136 
137     private static void
    testTwoPairOfSameDiffHighCard() {
138         ArrayList<Card> pokerHand = new ArrayList<
    Card>(Arrays.asList(
139             new Card(8, "Spades"),
140             new Card(8, "Hearts"),
141             new Card(7, "Diamonds"),
142             new Card(7, "Clubs"),
143             new Card(10, "Spades")
144         ));
145         ArrayList<Card> otherPokerHand = new
    ArrayList<Card>(Arrays.asList(
146             new Card(8, "Diamonds"),
147             new Card(8, "Clubs"),
148             new Card(7, "Spades"),
149             new Card(7, "Hearts"),
150             new Card(11, "Diamonds")
151         ));
152         PokerHand pokerHandObj = new PokerHand(
    pokerHand);
153         PokerHand otherPokerHandObj = new
    PokerHand(otherPokerHand);
154         int expected = -1;
155         int actual = pokerHandObj.compareTo(
    otherPokerHandObj);
156         Testing.assertEquals("Testing two pairs
    with same pairs but different high cards",
    expected, actual);
```

```
157     }
158 
159     private static void test9HighFlushv6HighFlush
    () {
160         ArrayList<Card> pokerHand = new ArrayList<
    Card>(Arrays.asList(
161             new Card(2, "Clubs"),
162             new Card(3, "Clubs"),
163             new Card(4, "Clubs"),
164             new Card(5, "Clubs"),
165             new Card(9, "Clubs")
166         ));
167         ArrayList<Card> otherPokerHand = new
    ArrayList<Card>(Arrays.asList(
168             new Card(4, "Spades"),
169             new Card(7, "Spades"),
170             new Card(8, "Spades"),
171             new Card(2, "Spades"),
172             new Card(6, "Spades")
173         ));
174         PokerHand pokerHandObj = new PokerHand(
    pokerHand);
175         PokerHand otherPokerHandObj = new
    PokerHand(otherPokerHand);
176         int expected = 1;
177         int actual = pokerHandObj.compareTo(
    otherPokerHandObj);
178         Testing.assertEquals("Testing 9-high flush
     vs 6-high flush", expected, actual);
179     }
180 
181     private static void
    testKing107FlushvKing105Flush() {
182         ArrayList<Card> pokerHand = new ArrayList<
    Card>(Arrays.asList(
183             new Card(10, "Hearts"),
184             new Card(7, "Hearts"),
185             new Card(3, "Hearts"),
186             new Card(2, "Hearts"),
187             new Card(4, "Hearts")
188         ));
```

```
189         ArrayList<Card> otherPokerHand = new
    ArrayList<Card>(Arrays.asList(
190             new Card(10, "Diamonds"),
191             new Card(5, "Diamonds"),
192             new Card(4, "Diamonds"),
193             new Card(2, "Diamonds"),
194             new Card(6, "Diamonds")
195         ));
196         PokerHand pokerHandObj = new PokerHand(
    pokerHand);
197         PokerHand otherPokerHandObj = new
    PokerHand(otherPokerHand);
198         int expected = 1;
199         int actual = pokerHandObj.compareTo(
    otherPokerHandObj);
200         Testing.assertEquals("Testing King-10-7
    flush vs King-10-5 flush", expected, actual);
201     }
202 
203     private static void testFlushActualTie() {
204         ArrayList<Card> pokerHand = new ArrayList<
    Card>(Arrays.asList(
205             new Card(2, "Hearts"),
206             new Card(4, "Hearts"),
207             new Card(6, "Hearts"),
208             new Card(8, "Hearts"),
209             new Card(10, "Hearts")
210         ));
211         ArrayList<Card> otherPokerHand = new
    ArrayList<Card>(Arrays.asList(
212             new Card(2, "Diamonds"),
213             new Card(4, "Diamonds"),
214             new Card(6, "Diamonds"),
215             new Card(8, "Diamonds"),
216             new Card(10, "Diamonds")
217         ));
218         PokerHand pokerHandObj = new PokerHand(
    pokerHand);
219         PokerHand otherPokerHandObj = new
    PokerHand(otherPokerHand);
220         int expected = 0;
```

```
221         int actual = pokerHandObj.compareTo(
    otherPokerHandObj);
222         Testing.assertEquals("Testing actual tie
    with flush", expected, actual);
223     }
224 
225     private static void testHighCardTie() {
226         ArrayList<Card> pokerHand = new ArrayList<
    Card>(Arrays.asList(
227             new Card(2, "Hearts"),
228             new Card(4, "Diamonds"),
229             new Card(6, "Spades"),
230             new Card(8, "Clubs"),
231             new Card(10, "Hearts")
232         ));
233         ArrayList<Card> otherPokerHand = new
    ArrayList<Card>(Arrays.asList(
234             new Card(2, "Clubs"),
235             new Card(4, "Hearts"),
236             new Card(6, "Diamonds"),
237             new Card(8, "Spades"),
238             new Card(10, "Clubs")
239         ));
240         PokerHand pokerHandObj = new PokerHand(
    pokerHand);
241         PokerHand otherPokerHandObj = new
    PokerHand(otherPokerHand);
242         int expected = 0;
243         int actual = pokerHandObj.compareTo(
    otherPokerHandObj);
244         Testing.assertEquals("Testing high card
    tie", expected, actual);
245     }
246 
247     private static void testTwoPairTie() {
248         ArrayList<Card> pokerHand = new ArrayList<
    Card>(Arrays.asList(
249             new Card(3, "Hearts"),
250             new Card(3, "Spades"),
251             new Card(6, "Diamonds"),
252             new Card(6, "Spades"),
```

```
253             new Card(10, "Clubs")
254         ));
255         ArrayList<Card> otherPokerHand = new
    ArrayList<Card>(Arrays.asList(
256             new Card(3, "Diamonds"),
257             new Card(3, "Clubs"),
258             new Card(6, "Hearts"),
259             new Card(6, "Clubs"),
260             new Card(10, "Spades")
261         ));
262         PokerHand pokerHandObj = new PokerHand(
    pokerHand);
263         PokerHand otherPokerHandObj = new
    PokerHand(otherPokerHand);
264         int expected = 0; // Assuming both hands
    have the same two pair
265         int actual = pokerHandObj.compareTo(
    otherPokerHandObj);
266         Testing.assertEquals("Testing two pair tie
    ", expected, actual);
267     }
268 
269     private static void testOnePairTie() {
270         ArrayList<Card> pokerHand = new ArrayList<
    Card>(Arrays.asList(
271             new Card(2, "Hearts"),
272             new Card(2, "Spades"),
273             new Card(4, "Diamonds"),
274             new Card(6, "Spades"),
275             new Card(10, "Clubs")
276         ));
277         ArrayList<Card> otherPokerHand = new
    ArrayList<Card>(Arrays.asList(
278             new Card(2, "Diamonds"),
279             new Card(2, "Clubs"),
280             new Card(4, "Hearts"),
281             new Card(6, "Clubs"),
282             new Card(10, "Spades")
283         ));
284         PokerHand pokerHandObj = new PokerHand(
    pokerHand);
```

```
285         PokerHand otherPokerHandObj = new
    PokerHand(otherPokerHand);
286         int expected = 0; // Assuming both hands
    have the same pair
287         int actual = pokerHandObj.compareTo(
    otherPokerHandObj);
288         Testing.assertEquals("Testing one pair tie
    ", expected, actual);
289     }
290 
291     private static void testFlushVHighCard() {
292         ArrayList<Card> pokerHand = new ArrayList<
    Card>(Arrays.asList(
293             new Card(2, "Hearts"),
294             new Card(4, "Hearts"),
295             new Card(6, "Hearts"),
296             new Card(8, "Hearts"),
297             new Card(10, "Hearts")
298         ));
299         ArrayList<Card> otherPokerHand = new
    ArrayList<Card>(Arrays.asList(
300             new Card(2, "Clubs"),
301             new Card(4, "Diamonds"),
302             new Card(6, "Spades"),
303             new Card(8, "Clubs"),
304             new Card(10, "Spades")
305         ));
306         PokerHand pokerHandObj = new PokerHand(
    pokerHand);
307         PokerHand otherPokerHandObj = new
    PokerHand(otherPokerHand);
308         int expected = 3;
309         int actual = pokerHandObj.compareTo(
    otherPokerHandObj);
310         Testing.assertEquals("Testing flush vs
    high card", expected, actual);
311     }
312 
313     private static void testTwoPairvOnePair() {
314         ArrayList<Card> pokerHand = new ArrayList<
    Card>(Arrays.asList(
```

```
315             new Card(5, "Hearts"),
316             new Card(5, "Spades"),
317             new Card(8, "Diamonds"),
318             new Card(8, "Spades"),
319             new Card(10, "Clubs")
320         ));
321         ArrayList<Card> otherPokerHand = new
    ArrayList<Card>(Arrays.asList(
322             new Card(5, "Diamonds"),
323             new Card(5, "Clubs"),
324             new Card(7, "Spades"),
325             new Card(9, "Hearts"),
326             new Card(3, "Diamonds")
327         ));
328         PokerHand pokerHandObj = new PokerHand(
    pokerHand);
329         PokerHand otherPokerHandObj = new
    PokerHand(otherPokerHand);
330         int expected = 1;
331         int actual = pokerHandObj.compareTo(
    otherPokerHandObj);
332         Testing.assertEquals("Testing two pair vs
    one pair", expected, actual);
333     }
334 
335     private static void testFullHousevStraight() {
336         ArrayList<Card> pokerHand = new ArrayList<
    Card>(Arrays.asList(
337             new Card(7, "Hearts"),
338             new Card(7, "Spades"),
339             new Card(7, "Diamonds"),
340             new Card(4, "Spades"),
341             new Card(4, "Clubs")
342         ));
343         ArrayList<Card> otherPokerHand = new
    ArrayList<Card>(Arrays.asList(
344             new Card(5, "Diamonds"),
345             new Card(6, "Clubs"),
346             new Card(7, "Spades"),
347             new Card(8, "Hearts"),
348             new Card(9, "Spades")
```

```
349         ));
350         PokerHand pokerHandObj = new PokerHand(
    pokerHand);
351         PokerHand otherPokerHandObj = new
    PokerHand(otherPokerHand);
352         int expected = 2;
353         int actual = pokerHandObj.compareTo(
    otherPokerHandObj);
354         Testing.assertEquals("Testing full house
    vs straight", expected, actual);
355     }
356 
357     private static void
    testRoyalFlushvStraightFlush() {
358         ArrayList<Card> pokerHand = new ArrayList<
    Card>(Arrays.asList(
359             new Card(10, "Hearts"),
360             new Card(11, "Hearts"),
361             new Card(12, "Hearts"),
362             new Card(13, "Hearts"),
363             new Card(14, "Hearts")
364         ));
365         ArrayList<Card> otherPokerHand = new
    ArrayList<Card>(Arrays.asList(
366             new Card(7, "Spades"),
367             new Card(8, "Spades"),
368             new Card(9, "Spades"),
369             new Card(10, "Spades"),
370             new Card(11, "Spades")
371         ));
372         PokerHand pokerHandObj = new PokerHand(
    pokerHand);
373         PokerHand otherPokerHandObj = new
    PokerHand(otherPokerHand);
374         int expected = 1;
375         int actual = pokerHandObj.compareTo(
    otherPokerHandObj);
376         Testing.assertEquals("Testing royal flush
    vs straight flush", expected, actual);
377     }
378 
```

```
379     private static void testFourOfAKindV2Pair() {
380         ArrayList<Card> pokerHand = new ArrayList<
    Card>(Arrays.asList(
381             new Card(6, "Hearts"),
382             new Card(6, "Spades"),
383             new Card(6, "Diamonds"),
384             new Card(6, "Clubs"),
385             new Card(9, "Clubs")
386         ));
387         ArrayList<Card> otherPokerHand = new
    ArrayList<Card>(Arrays.asList(
388             new Card(8, "Diamonds"),
389             new Card(8, "Clubs"),
390             new Card(7, "Spades"),
391             new Card(7, "Hearts"),
392             new Card(3, "Spades")
393         ));
394         PokerHand pokerHandObj = new PokerHand(
    pokerHand);
395         PokerHand otherPokerHandObj = new
    PokerHand(otherPokerHand);
396         int expected = -1; // Assuming the second
    hand has two pairs
397         int actual = pokerHandObj.compareTo(
    otherPokerHandObj);
398         Testing.assertEquals("Testing four of a
    kind vs two pair", expected, actual);
399     }
400 
401     private static void
    testHighCardTieUpToThirdCard() {
402         ArrayList<Card> pokerHand = new ArrayList<
    Card>(Arrays.asList(
403             new Card(2, "Hearts"),
404             new Card(4, "Diamonds"),
405             new Card(6, "Spades"),
406             new Card(8, "Clubs"),
407             new Card(10, "Hearts")
408         ));
409         ArrayList<Card> otherPokerHand = new
    ArrayList<Card>(Arrays.asList(
```

```
410             new Card(2, "Clubs"),
411             new Card(4, "Hearts"),
412             new Card(6, "Diamonds"),
413             new Card(9, "Spades"),
414             new Card(10, "Clubs")
415         ));
416         PokerHand pokerHandObj = new PokerHand(
    pokerHand);
417         PokerHand otherPokerHandObj = new
    PokerHand(otherPokerHand);
418         int expected = -1;
419         int actual = pokerHandObj.compareTo(
    otherPokerHandObj);
420         Testing.assertEquals("Testing high card
    tie up to the third card", expected, actual);
421     }
422 
423     private static void
    testOnePairTieTillFourthCard() {
424         ArrayList<Card> pokerHand = new ArrayList<
    Card>(Arrays.asList(
425             new Card(2, "Hearts"),
426             new Card(2, "Spades"),
427             new Card(4, "Diamonds"),
428             new Card(6, "Spades"),
429             new Card(10, "Clubs")
430         ));
431         ArrayList<Card> otherPokerHand = new
    ArrayList<Card>(Arrays.asList(
432             new Card(2, "Diamonds"),
433             new Card(2, "Clubs"),
434             new Card(4, "Hearts"),
435             new Card(6, "Clubs"),
436             new Card(9, "Spades")
437         ));
438         PokerHand pokerHandObj = new PokerHand(
    pokerHand);
439         PokerHand otherPokerHandObj = new
    PokerHand(otherPokerHand);
440         int expected = 1;
441         int actual = pokerHandObj.compareTo(
```

```
441 otherPokerHandObj);
442         Testing.assertEquals("Testing one pair tie
     up to the fourth card", expected, actual);
443     }
444 
445     private static void testThreeOfAKindVsTwoPair
    () {
446         ArrayList<Card> pokerHand = new ArrayList<
    Card>(Arrays.asList(
447             new Card(7, "Hearts"),
448             new Card(7, "Spades"),
449             new Card(7, "Diamonds"),
450             new Card(4, "Spades"),
451             new Card(4, "Clubs")
452         ));
453         ArrayList<Card> otherPokerHand = new
    ArrayList<Card>(Arrays.asList(
454             new Card(5, "Diamonds"),
455             new Card(5, "Clubs"),
456             new Card(7, "Spades"),
457             new Card(7, "Hearts"),
458             new Card(9, "Spades")
459         ));
460         PokerHand pokerHandObj = new PokerHand(
    pokerHand);
461         PokerHand otherPokerHandObj = new
    PokerHand(otherPokerHand);
462         int expected = -1;
463         int actual = pokerHandObj.compareTo(
    otherPokerHandObj);
464         Testing.assertEquals("Testing three of a
    kind vs two pair", expected, actual);
465     }
466 
467 
468 
469 }
470 
```