```java
/**
 * Models a single playing card
 */

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

public class Card {

    private final int JACK = 11;
    private final int QUEEN = 12;
    private final int KING = 13;
    private final int ACE = 14;
    private int rank;
    private String suit;

    /**
     * Non-default constructor for Card
     * @param rank an integer for the rank
     * @param suit a string for the suit
     */
    public Card(int rank, String suit) {
        this.rank = rank;
        this.suit = suit;
    }

    /**
     * Getter for the rank
     * @return an integer for the rank
     */
    public int getRank() {return rank;}

    /**
     * Getter for the suit
     * @return a string for the suit
     */
    public String getSuit() {return suit;}

    /**
     * Gets the string version of the rank
     * @return a string for the rank
     */
```

```
42      public String getStringRank() {
43          int currentRank = getRank();
44          String stringRank;
45
46          if (currentRank == JACK) {
47              stringRank = "Jack";
48          }
49
50          else if (currentRank == QUEEN) {
51              stringRank = "Queen";
52          }
53
54          else if (currentRank == KING) {
55              stringRank = "King";
56          }
57
58          else if (currentRank == ACE) {
59              stringRank = "Ace";
60          }
61
62          else {
63              stringRank = String.valueOf(currentRank);
64          }
65
66          return stringRank;
67      }
68
69      /**
70       * Returns the readable version of the card
71       * @return a string for the readable version of the card
72       */
73      public String toString() {
74          return getStringRank() + " of " + getSuit();
75      }
76  }
```

```java
/**
 * Models a deck of cards
 */

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

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

public class Deck {

    private final int START = 0;
    private final int[] RANKS = new int[] {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14};
    private final String[] SUITS = new String[] {"Hearts", "Diamonds", "Spades", "Clubs"};
    private final int MAX_DECK = 52;
    private final int EMPTY = 0;
    private ArrayList<Card> deck = new ArrayList<Card>();
    private int nextToDeal;

    /**
     * Default constructor for the deck
     */
    public Deck() {
        for (int rank : RANKS) {
            for (String suit : SUITS) {
                deck.add(new Card(rank, suit));
            }
        }

        nextToDeal = START;
    }

    /**
     * Shuffles the deck
     */
    public void shuffle() {
        for (int i = nextToDeal; i < MAX_DECK; i++) {
            int randomNumber = ThreadLocalRandom.current().nextInt(nextToDeal, MAX_DECK);
```

```
39              Card currentCard = deck.get(i);
40              Card randomCard = deck.get(randomNumber);
41
42              deck.set(i, randomCard);
43              deck.set(randomNumber, currentCard);
44          }
45      }
46
47      /**
48       * Returns the next undealt card or null if deck is empty
49       * @return the next undealt card or null if deck is empty
50       */
51      public Card deal(){
52          if (deck.isEmpty()) {
53              return null;
54          }
55
56          Card currentCard = deck.get(nextToDeal);
57
58          nextToDeal++;
59
60          return currentCard;
61      }
62
63      /**
64       * Checks whether the deck is empty
65       * @return true if there are still undealt cards in the
   deck
66       */
67      public boolean isEmpty() {
68          if (size() == EMPTY){
69              return true;
70          }
71
72          else {
73              return false;
74          }
75      }
76
77      /**
78       * Returns the number of undealt cards in the deck
```

```java
     * @return the number of undealt cards in the deck
     */
    public int size() {return MAX_DECK - nextToDeal;}

    /**
     * Returns the deck to a state where all cards are
  undealt in a shuffled state
     */
    public void gather() {nextToDeal = START;}

    /**
     * Returns all the undealt cards in the deck as a string
     * @return all the undealt cards in the deck as a string
     */
    public String toString() {
        String returnString = "";

        for (int i = nextToDeal; i < MAX_DECK; i++) {
            returnString += deck.get(i) + "\n";
        }

        return returnString;
    }

    /**
     * Deals the card i times
     * @return an array list for the card list
     */
    public ArrayList dealITimes(int i){
        ArrayList<Card> cardList = new ArrayList<Card>();

        for (int j = START; j < i; j++) {
            cardList.add(deal());
        }

        return cardList;
    }
}
```

```java
/**
 * A simple poker game
 */

package proj3;

import java.util.Scanner;

public class Client {

    private final boolean CONTINUE = true;
    private final int MAX_HAND = 5;

    public static void main(String[] args) {
        Client client = new Client();
        Deck deck = new Deck();
        deck.shuffle();
        boolean continueGame = client.CONTINUE;
        int totalPoint = 0;

        while (continueGame && deck.size() > client.MAX_HAND * 2) {
            PokerHand myHand = new PokerHand(deck.dealITimes(client.MAX_HAND));
            PokerHand otherHand = new PokerHand(deck.dealITimes(client.MAX_HAND));
            String result = myHand.getResult(otherHand);

            System.out.println("my hand: " + myHand);
            System.out.println("other hand: " + otherHand);

            Scanner sc = new Scanner(System.in);
            System.out.println("Who is the winner? (Type my hand, other hand, or tie)");

            if (sc.nextLine().equals(result)) {
                totalPoint++;
            }

            else {
                continueGame = ! client.CONTINUE;
```

```
38              }
39          }
40
41          System.out.println("Game is over, and your total point
    is " + totalPoint);
42      }
43  }
```

```java
package proj3;

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

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

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

    /**
     * Each of the assertEquals methods tests whether the actual
     * result equals the expected result. If it does, then the test
```

```
35      * passes, otherwise it fails.
36      *
37      * The only difference between these methods is the types
    of the
38      * parameters.
39      *
40      * All take a String message and two values of some other
    type to
41      * compare:
42      *
43      * @param message
44      *            a message or description of the test
45      * @param expected
46      *            the correct, or expected, value
47      * @param actual
48      *            the actual value
49      */
50     public static void assertEquals(String message, boolean
    expected,
51                                     boolean actual)
52     {
53         printTestCaseInfo(message, "" + expected, "" + actual
    );
54         if (expected == actual) {
55             pass();
56         } else {
57             fail(message);
58         }
59     }
60
61     public static void assertEquals(String message, int
    expected, int actual)
62     {
63         printTestCaseInfo(message, "" + expected, "" + actual
    );
64         if (expected == actual) {
65             pass();
66         } else {
67             fail(message);
68         }
69     }
```

```
 70
 71     public static void assertEquals(String message, Object
    expected,
 72                                     Object actual)
 73     {
 74         String expectedString = "<<null>>";
 75         String actualString = "<<null>>";
 76         if (expected != null) {
 77             expectedString = expected.toString();
 78         }
 79         if (actual != null) {
 80             actualString = actual.toString();
 81         }
 82         printTestCaseInfo(message, expectedString,
    actualString);
 83
 84         if (expected == null) {
 85             if (actual == null) {
 86                 pass();
 87             } else {
 88                 fail(message);
 89             }
 90         } else if (expected.equals(actual)) {
 91             pass();
 92         } else {
 93             fail(message);
 94         }
 95     }
 96
 97     /**
 98      * Asserts that a given boolean must be true.  The test
    fails if
 99      * the boolean is not true.
100      *
101      * @param message The test message
102      * @param actual The boolean value asserted to be true.
103      */
104     public static void assertTrue(String message, boolean
    actual)
105     {
106         assertEquals(message, true, actual);
```

```
107     }
108
109     /**
110      * Asserts that a given boolean must be false. The test
    fails if
111      * the boolean is not false (i.e. if it is true).
112      *
113      * @param message The test message
114      * @param actual The boolean value asserted to be false.
115      */
116     public static void assertFalse(String message, boolean
    actual)
117     {
118         assertEquals(message, false, actual);
119     }
120
121     private static void printTestCaseInfo(String message,
    String expected,
122                                           String actual)
123     {
124         if (VERBOSE) {
125             System.out.println(message + ":");
126             System.out.println("expected: " + expected);
127             System.out.println("actual:   " + actual);
128         }
129     }
130
131     private static void pass()
132     {
133         numTests++;
134
135         if (VERBOSE) {
136             System.out.println("--PASS--");
137             System.out.println();
138         }
139     }
140
141     private static void fail(String description)
142     {
143         numTests++;
144         numFails++;
```

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

```
184 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
/**
 * Models a 5-card hand of cards
 */

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

import java.util.ArrayList;
import java.util.Comparator;

public class PokerHand {

    private final int MAX_HAND = 5;
    private final int TIE = 0;
    private final int START = 0;
    private final int FLUSH = 4;
    private final int TWO_PAIR = 3;
    private final int PAIR = 2;
    private final int HIGH_CARD = 1;
    private final int FIRST_CARD = 0;
    private final int LAST_CARD = 1;

    private ArrayList<Card> hand;
    private ArrayList<Integer> rankList;

    /**
     * Non-default constructor for the hand
     * @param cardList a list of cards that should be in the hand
     */
    public PokerHand(ArrayList<Card> cardList) {
        hand = new ArrayList<Card>(cardList);
    }

    /**
     * Adds the card to the hand if the hand does not have 5 cards in it
     * @param card a Card object that will be added to the hand
     */
    public void addCard(Card card) {
        if (hand.size() < MAX_HAND) {
```

```
39              hand.add(card);
40          }
41      }
42
43      /**
44       * Getter for the card at the given index
45       * @param index an integer greater or equal to 0
46       * @return a card object at the given index or null if
   index is invalid
47       */
48      public Card getIthCard(int index){
49          if (index < hand.size()) {
50              return hand.get(index);
51          }
52
53          else {
54              return null;
55          }
56      }
57
58      /**
59       * Returns the readable version of the hand
60       * @return a string for the readable version of the hand
61       */
62      public String toString(){
63          String returnString = "";
64
65          for (Card card : hand) {
66              returnString += card + "\n";
67          }
68
69          return returnString;
70      }
71
72      /**
73       * Determines how this hand compares to another hand,
   returns
74       * positive, negative, or zero depending on the
   comparision.
75       *
76       * @param other The hand to compare this hand to
```

```
77      * @return a negtive number if this is worth LESS than
   other, zero
78      * if they are worth the SAME, and a positive number if
   this is worth
79      * MORE than other
80      */
81     public int compareTo(PokerHand other){
82         int myPoint = getPoint();
83         int otherPoint = other.getPoint();
84 
85         int result = myPoint - otherPoint;
86 
87         if (result == TIE) {
88             if (myPoint == FLUSH || myPoint == HIGH_CARD) {
89                 getRankList();
90                 other.getRankList();
91 
92                 for (int i = START; i < MAX_HAND; i++) {
93                     result = getIthRank(i) - other.getIthRank
   (i);
94 
95                     if (result != TIE) {
96                         return result;
97                     }
98                 }
99             }
100 
101             else if (myPoint == TWO_PAIR || myPoint == PAIR
   ) {
102                 for (int i = START; i < rankListSize(); i
   ++) {
103                     result = getIthRank(i) - other.getIthRank
   (i);
104 
105                     if (result != TIE) {
106                         return result;
107                     }
108                 }
109             }
110 
111             else {
```

```
112                 System.out.println("These hands are invalid"
    );
113             }
114         }
115 
116         return result;
117     }
118 
119     /** Gets the point of the hand
120      * @return an integer for the point of the hand
121      */
122     private int getPoint(){
123         int handScore;
124 
125         if (isFlush()){
126             handScore = FLUSH;
127         }
128 
129         else if (isTwoPair()){
130             handScore = TWO_PAIR;
131         }
132 
133         else if (isPair()){
134             handScore = PAIR;
135         }
136 
137         else {
138             handScore = HIGH_CARD;
139         }
140 
141         return handScore;
142     }
143 
144     /**
145      * Gets the list with ranks in the descending order
146      */
147     private void getRankList() {
148         ArrayList<Integer> rankList = new ArrayList<Integer
    >();
149 
150         for (Card card : hand) {
```

```
151             rankList.add(card.getRank());
152         }
153 
154         rankList.sort(Comparator.reverseOrder());
155 
156         this.rankList = rankList;
157     }
158 
159     /**
160      * Getter for the rank at the given index
161      * @param index an integer for the index of the rank
162      * @rank an integer at the given index
163      */
164     private int getIthRank(int index){return rankList.get(
    index);}
165 
166     /**
167      * Returns the size of the rank list
168      * @return an integer for the size of the rank list
169      */
170     private int rankListSize(){return rankList.size();}
171 
172     /**
173      * Returns the number of cards left in the hand
174      * @return an integer for cards left in the hand
175      */
176     private int size(){return hand.size();}
177 
178     /**
179      * Removes the card with the given index
180      * @param index an integer for the index of the card
181      */
182     private void removeIthCard(int index) {
183         if (index < hand.size()) {
184             hand.remove(index);
185         }
186     }
187 
188     /**
189      * Checks whether the hand is flush
190      * @return true if all cards have the same suit
```

```
191      */
192     private boolean isFlush(){
193         for (int i = 1; i < MAX_HAND; i++) {
194             if (getIthCard(i).getSuit() != getIthCard(i - 1).
    getSuit()) {
195                 return false;
196             }
197         }
198 
199         return true;
200     }
201 
202     /**
203      * Checks whether the hand is two pair
204      * @return true if the hand has 2 pairs of the same rank
205      */
206     private boolean isTwoPair(){
207         PokerHand currentHand = new PokerHand(hand);
208 
209         int i = 1;
210         int totalPair = 0;
211         ArrayList<Integer> pairRankList = new ArrayList<
    Integer>();
212         ArrayList<Integer> otherRankList = new ArrayList<
    Integer>();
213 
214         while (currentHand.size() > i) {
215             if (currentHand.getIthCard(FIRST_CARD).getRank
    () == currentHand.getIthCard(i).getRank()) {
216                 totalPair++;
217 
218                 pairRankList.add(currentHand.getIthCard(
    FIRST_CARD).getRank());
219 
220                 currentHand.removeIthCard(i);
221                 currentHand.removeIthCard(FIRST_CARD);
222 
223                 i = 1;
224             }
225 
226             else {
```

```
227                 i++;
228             }
229 
230             if (currentHand.size() == i) {
231                 otherRankList.add(currentHand.getIthCard(
    FIRST_CARD).getRank());
232 
233                 currentHand.removeIthCard(FIRST_CARD);
234 
235                 i = 1;
236             }
237 
238             if (currentHand.size() == LAST_CARD) {
239                 otherRankList.add(currentHand.getIthCard(
    FIRST_CARD).getRank());
240 
241                 currentHand.removeIthCard(FIRST_CARD);
242             }
243         }
244         if (totalPair == 2) {
245             pairRankList.sort(Comparator.reverseOrder());
246             otherRankList.sort(Comparator.reverseOrder());
247 
248             pairRankList.addAll(otherRankList);
249 
250             this.rankList = pairRankList;
251 
252             return true;
253         }
254 
255         return false;
256     }
257     /**
258      * Checks whether the hand is a pair
259      * @return true if the hand has a pair of the same rank
260      */
261     private boolean isPair(){
262         PokerHand currentHand = new PokerHand(hand);
263 
264         int i = 1;
265         int totalPair = 0;
```

```
266         ArrayList<Integer> pairRankList = new ArrayList<
    Integer>();
267         ArrayList<Integer> otherRankList = new ArrayList<
    Integer>();
268 
269         while (currentHand.size() > i) {
270             if (currentHand.getIthCard(FIRST_CARD).getRank
    () == currentHand.getIthCard(i).getRank()) {
271                 totalPair++;
272 
273                 pairRankList.add(currentHand.getIthCard(
    FIRST_CARD).getRank());
274 
275                 currentHand.removeIthCard(i);
276                 currentHand.removeIthCard(FIRST_CARD);
277 
278                 i = 1;
279             }
280 
281             else {
282                 i++;
283             }
284 
285             if (currentHand.size() == i) {
286                 otherRankList.add(currentHand.getIthCard(
    FIRST_CARD).getRank());
287 
288                 currentHand.removeIthCard(FIRST_CARD);
289 
290                 i = 1;
291             }
292 
293             if (currentHand.size() == LAST_CARD) {
294                 otherRankList.add(currentHand.getIthCard(
    FIRST_CARD).getRank());
295 
296                 currentHand.removeIthCard(FIRST_CARD);
297             }
298         }
299 
300         if (totalPair == 1) {
```

```
301             pairRankList.sort(Comparator.reverseOrder());
302             otherRankList.sort(Comparator.reverseOrder());
303 
304             pairRankList.addAll(otherRankList);
305 
306             this.rankList = pairRankList;
307 
308             return true;
309         }
310 
311         return false;
312     }
313 
314     /**
315      * Gets the result according to the given value
316      * @return a string according to the given value
317      */
318     public String getResult(PokerHand other) {
319         int result = compareTo(other);
320 
321         if (result > 0) {
322             return "my hand";
323         }
324 
325         else if (result < 0) {
326             return "other hand";
327         }
328 
329         else {
330             return "tie";
331         }
332     }
333 }
```

```java
package proj3;

import java.lang.reflect.Array;

public class CardTester {

    private final int START = 0;
    private final int[] INTEGER_RANKS = new int[] {11, 12, 13, 14};
    private final String[] STRING_RANKS = new String[] {"Jack", "Queen", "King", "Ace"};

    public static void main(String[] args) {
        CardTester cardTester = new CardTester();

        Testing.startTests();
        cardTester.testCardConstructor();
        cardTester.testGetRank();
        cardTester.testGetSuit();
        cardTester.testGetStringRank();
        Testing.finishTests();
    }

    private void testCardConstructor() {
        Card card = new Card(2, "Hearts");
        String msg = "Starts testing card constructor and toString method";
        String expected = "2 of Hearts";
        String actual = card.toString();

        Testing.assertEquals(msg, expected, actual);
    }

    private void testGetRank() {
        Card card = new Card(2, "Hearts");
        String msg = "Starts testing getRank method";
        int expected = 2;
        int actual = card.getRank();

        Testing.assertEquals(msg, expected, actual);
    }
```

```
39
40      private void testGetSuit() {
41          Card card = new Card(2, "Hearts");
42          String msg = "Starts testing getSuit method";
43          String expected = "Hearts";
44          String actual = card.getSuit();
45
46          Testing.assertEquals(msg, expected, actual);
47      }
48
49      private void testGetStringRank() {
50          for (int i = START; i < INTEGER_RANKS.length; i++) {
51              Card card = new Card((int) Array.get(INTEGER_RANKS
    , i), "Hearts");
52              String expected = Array.get(STRING_RANKS, i).
    toString();
53              String msg = "Start testing string rank " +
    expected;
54              String actual = card.getStringRank();
55
56              Testing.assertEquals(msg, expected, actual);
57          }
58      }
59  }
```

```java
package proj3;

import java.util.ArrayList;
import java.util.Random;

public class DeckTester {

    private final int START = 0;
    private final int MAX_DECK = 52;

    public static void main(String[] args) {
        DeckTester deckTester = new DeckTester();

        Testing.startTests();
        deckTester.testShuffle();
        deckTester.testDeal();
        deckTester.testIsEmpty();
        deckTester.testSize();
        deckTester.testGather();
        deckTester.testDealITimes();
        Testing.finishTests();
    }

    private void testDeckConstructor() {
        Deck deck = new Deck();
        String msg = "Starts testing deck constructor and toString method";
        String expected = "2 of Hearts\n" + "2 of Diamonds\n" + "2 of Spades\n" + "2 of Clubs\n" + "3 of Hearts\n"
                + "3 of Diamonds\n" + "3 of Spades\n" + "3 of Clubs\n" + "4 of Hearts\n" + "4 of Diamonds\n" + "4 of Spades\n"
                + "4 of Clubs\n" + "5 of Hearts\n" + "5 of Diamonds\n" + "5 of Spades\n" + "5 of Clubs\n" + "6 of Hearts\n"
                + "6 of Diamonds\n" + "6 of Spades\n" + "6 of Clubs\n" + "7 of Hearts\n" + "7 of Diamonds\n" + "7 of Spades\n"
                + "7 of Clubs\n" + "8 of Hearts\n" + "8 of Diamonds\n" + "8 of Spades\n" + "8 of Clubs\n" + "9 of Hearts\n"
```

```
32                  + "9 of Diamonds\n" + "9 of Spades\n" + "9 of
   Clubs\n" + "10 of Hearts\n" + "10 of Diamonds\n" + "10 of
   Spades\n"
33                  + "10 of Clubs\n" + "Jack of Hearts\n" + "Jack
    of Diamonds\n" + "Jack of Spades\n" + "Jack of Clubs\n"
34                  + "Queen of Hearts\n" + "Queen of Diamonds\n"
    + "Queen of Spades\n" + "Queen of Clubs\n" + "King of Hearts\
   n"
35                  + "King of Diamonds\n" + "King of Spades\n" +
   "King of Clubs\n" + "Ace of Hearts\n" + "Ace of Diamonds\n"
36                  + "Ace of Spades\n" + "Ace of Clubs";
37          Deck actual = deck;
38
39          Testing.assertEquals(msg, expected, actual.toString
   ());
40      }
41
42      private void testShuffle() {
43          Deck deck = new Deck();
44          Deck shuffledDeck = new Deck();
45          shuffledDeck.shuffle();
46          String msg = "Start testing shuffle method";
47          String expected = "Shuffled";
48          String actual = "Shuffled";
49
50          int totalMatch = 0;
51
52          for (int i = START; i < MAX_DECK; i++) {
53              if (deck.deal().toString().equals(shuffledDeck.
   deal().toString())) {
54                  totalMatch++;
55              }
56          }
57
58          if (totalMatch > MAX_DECK / 2) {
59              actual = "Not shuffled";
60          }
61
62          Testing.assertEquals(msg, expected, actual);
63      }
64
```

```
65      private void testDeal() {
66          Deck deck = new Deck();
67          String msg = "Starts testing deal method";
68          Card expected = new Card(2, "Hearts");
69          Card actual = deck.deal();
70
71          Testing.assertEquals(msg, expected.toString(), actual
    .toString());
72      }
73
74      private void testIsEmpty() {
75          testNotEmptyDeck();
76          testEmptyDeck();
77      }
78
79      private void testNotEmptyDeck() {
80          Deck deck = new Deck();
81          String msg = "Starts testing not empty deck";
82          boolean expected = false;
83          boolean actual = deck.isEmpty();
84
85          Testing.assertEquals(msg, expected, actual);
86      }
87
88      private void testEmptyDeck() {
89          Deck deck = new Deck();
90
91          for (int i = START; i < MAX_DECK; i++) {
92              deck.deal();
93          }
94
95          String msg = "Starts testing empty deck";
96          boolean expected = true;
97          boolean actual = deck.isEmpty();
98
99          Testing.assertEquals(msg, expected, actual);
100     }
101
102     private void testSize() {
103         Deck deck = new Deck();
104         String msg = "Starts testing size method";
```

```
105         int expected = MAX_DECK;
106 
107         for (int i = START; i < MAX_DECK; i++) {
108             deck.deal();
109             expected--;
110 
111             int actual = deck.size();
112 
113             Testing.assertEquals(msg, expected, actual);
114         }
115     }
116 
117     private void testGather() {
118         Deck deck = new Deck();
119         Random randomNumber = new Random();
120         deck.dealITimes(randomNumber.nextInt(MAX_DECK));
121         deck.gather();
122 
123         String msg = "Starts testing gather method";
124         int expected = MAX_DECK;
125         int actual = deck.size();
126 
127         Testing.assertEquals(msg, expected, actual);
128     }
129 
130     private void testDealITimes() {
131         Deck deck = new Deck();
132         String msg = "Starts testing dealITimes method";
133         ArrayList<Card> expected = new ArrayList<Card>();
134 
135         expected.add(new Card(2, "Hearts"));
136         expected.add(new Card(2, "Diamonds"));
137         expected.add(new Card(2, "Spades"));
138         expected.add(new Card(2, "Clubs"));
139         expected.add(new Card(3, "Hearts"));
140 
141         ArrayList<Card> actual = deck.dealITimes(5);
142 
143         Testing.assertEquals(msg, expected.toString(), actual
    .toString());
144     }
```

```
145 }
```

```java
/**
 * @author Chris Hegang Kim
 * @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.
 */

package proj3;

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

public class PokerComparisionTests {

    public static void main(String[] args) {
        PokerComparisionTests handTester = new PokerComparisionTests();

        Testing.startTests();
        handTester.testCompareTo();
        Testing.finishTests();
    }

    private void testCompareTo() {
        flushVsTwoPair();
        flushVsFlushHighCard1();
        flushVsFlushHighCard2();
        flushVsFlushHighCard3();
        flushVsFlushHighCard4();
        flushVsFlushHighCard5();
        flushVsFlushTie();

        twoPairVsPair();
        twoPairVsTwoPairHighCard1();
        twoPairVsTwoPairHighCard2();
        twoPairVsTwoPairHighCard3();
        twoPairVsTwoPairTie();

        pairVsHighCard();
        pairVsPairHighCard1();
```

```
39          pairVsPairHighCard2();
40          pairVsPairHighCard3();
41          pairVsPairHighCard4();
42          pairVsPairTie();
43
44          highCardVsHighCard1();
45          highCardVsHighCard2();
46          highCardVsHighCard3();
47          highCardVsHighCard4();
48          highCardVsHighCard5();
49          highCardVsHighCardTie();
50      }
51
52      private void flushVsTwoPair() {
53          PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
54                  new Card(14, "Hearts"), new Card(8, "Hearts"
    ), new Card(7, "Hearts"),
55                  new Card(10, "Hearts"))));
56          PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
57                  new Card(14, "Diamonds"), new Card(8, "Spades"
    ), new Card(8, "Clubs"),
58                  new Card(9, "Hearts"))));
59          String msg = "Start testing flush vs two pair";
60          int expected = 1;
61          int actual = hand.compareTo(other);
62
63          Testing.assertEquals(msg, expected, actual);
64      }
65
66      private void flushVsFlushHighCard1() {
67          PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
68                  new Card(13, "Hearts"), new Card(12, "Hearts"
    ), new Card(11, "Hearts"),
69                  new Card(10, "Hearts"))));
70          PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(9, "Hearts"),
71                  new Card(13, "Hearts"), new Card(12, "Hearts"
    ), new Card(11, "Hearts"),
```

```
72                  new Card(10, "Hearts"))));
73          String msg = "Start testing flush vs flush (first
    high card)";
74          int expected = 1;
75          int actual = hand.compareTo(other);
76
77          Testing.assertEquals(msg, expected, actual);
78      }
79
80      private void flushVsFlushHighCard2() {
81          PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
82                  new Card(13, "Hearts"), new Card(12, "Hearts"
    ), new Card(11, "Hearts"),
83                  new Card(10, "Hearts"))));
84          PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(9, "Hearts"),
85                  new Card(14, "Hearts"), new Card(12, "Hearts"
    ), new Card(11, "Hearts"),
86                  new Card(10, "Hearts"))));
87          String msg = "Start testing flush vs flush (second
    high card)";
88          int expected = 1;
89          int actual = hand.compareTo(other);
90
91          Testing.assertEquals(msg, expected, actual);
92      }
93
94      private void flushVsFlushHighCard3() {
95          PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
96                  new Card(13, "Hearts"), new Card(12, "Hearts"
    ), new Card(11, "Hearts"),
97                  new Card(10, "Hearts"))));
98          PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(9, "Hearts"),
99                  new Card(14, "Hearts"), new Card(13, "Hearts"
    ), new Card(11, "Hearts"),
100                 new Card(10, "Hearts"))));
101         String msg = "Start testing flush vs flush (third
    high card)";
```

```
102         int expected = 1;
103         int actual = hand.compareTo(other);
104 
105         Testing.assertEquals(msg, expected, actual);
106     }
107 
108     private void flushVsFlushHighCard4() {
109         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
110                 new Card(13, "Hearts"), new Card(12, "Hearts"
    ), new Card(11, "Hearts"),
111                 new Card(10, "Hearts"))));
112         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(9, "Hearts"),
113                 new Card(14, "Hearts"), new Card(13, "Hearts"
    ), new Card(12, "Hearts"),
114                 new Card(10, "Hearts"))));
115         String msg = "Start testing flush vs flush (fourth
    high card)";
116         int expected = 1;
117         int actual = hand.compareTo(other);
118 
119         Testing.assertEquals(msg, expected, actual);
120     }
121 
122     private void flushVsFlushHighCard5() {
123         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
124                 new Card(13, "Hearts"), new Card(12, "Hearts"
    ), new Card(11, "Hearts"),
125                 new Card(10, "Hearts"))));
126         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(9, "Hearts"),
127                 new Card(14, "Hearts"), new Card(13, "Hearts"
    ), new Card(12, "Hearts"),
128                 new Card(11, "Hearts"))));
129         String msg = "Start testing flush vs flush (fifth
    high card)";
130         int expected = 1;
131         int actual = hand.compareTo(other);
132 
```

```
133         Testing.assertEquals(msg, expected, actual);
134     }
135 
136     private void flushVsFlushTie() {
137         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
138                 new Card(13, "Hearts"), new Card(12, "Hearts"
    ), new Card(11, "Hearts"),
139                 new Card(10, "Hearts"))));
140         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(10, "Hearts"),
141                 new Card(14, "Hearts"), new Card(13, "Hearts"
    ), new Card(12, "Hearts"),
142                 new Card(11, "Hearts"))));
143         String msg = "Start testing flush vs flush (tie)";
144         int expected = 0;
145         int actual = hand.compareTo(other);
146 
147         Testing.assertEquals(msg, expected, actual);
148     }
149 
150     private void twoPairVsPair() {
151         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
152                 new Card(14, "Diamonds"), new Card(8, "Spades
    "), new Card(8, "Clubs"),
153                 new Card(10, "Hearts"))));
154         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
155                 new Card(14, "Diamonds"), new Card(8, "Spades
    "), new Card(9, "Clubs"),
156                 new Card(10, "Hearts"))));
157         String msg = "Start testing two pair vs pair";
158         int expected = 1;
159         int actual = hand.compareTo(other);
160 
161         Testing.assertEquals(msg, expected, actual);
162     }
163 
164     private void twoPairVsTwoPairHighCard1() {
165         PokerHand hand = new PokerHand(new ArrayList<Card>(
```

```
165 Arrays.asList(new Card(14, "Hearts"),
166                 new Card(14, "Diamonds"), new Card(8, "Hearts
    "), new Card(8, "Hearts"),
167                 new Card(9, "Hearts"))));
168         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(13, "Hearts"),
169                 new Card(13, "Diamonds"), new Card(11, "
    Hearts"), new Card(11, "Hearts"),
170                 new Card(9, "Hearts"))));
171         String msg = "Start testing two pair vs two pair (
    first high pair card)";
172         int expected = 1;
173         int actual = hand.compareTo(other);
174 
175         Testing.assertEquals(msg, expected, actual);
176     }
177 
178     private void twoPairVsTwoPairHighCard2() {
179         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
180                 new Card(14, "Diamonds"), new Card(8, "Hearts
    "), new Card(8, "Hearts"),
181                 new Card(9, "Hearts"))));
182         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
183                 new Card(14, "Diamonds"), new Card(7, "Hearts
    "), new Card(7, "Hearts"),
184                 new Card(9, "Hearts"))));
185         String msg = "Start testing two pair vs two pair (
    second high pair card)";
186         int expected = 1;
187         int actual = hand.compareTo(other);
188 
189         Testing.assertEquals(msg, expected, actual);
190     }
191 
192     private void twoPairVsTwoPairHighCard3() {
193         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
194                 new Card(14, "Diamonds"), new Card(8, "Hearts
    "), new Card(8, "Hearts"),
```

```
195                 new Card(10, "Hearts"))));
196         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
197                 new Card(14, "Diamonds"), new Card(8, "Hearts
    "), new Card(8, "Hearts"),
198                 new Card(9, "Hearts"))));
199         String msg = "Start testing two pair vs two pair (
    third high card)";
200         int expected = 1;
201         int actual = hand.compareTo(other);
202 
203         Testing.assertEquals(msg, expected, actual);
204     }
205 
206     private void twoPairVsTwoPairTie() {
207         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
208                 new Card(14, "Diamonds"), new Card(8, "Hearts
    "), new Card(8, "Hearts"),
209                 new Card(10, "Hearts"))));
210         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
211                 new Card(14, "Diamonds"), new Card(8, "Hearts
    "), new Card(8, "Hearts"),
212                 new Card(10, "Hearts"))));
213         String msg = "Start testing two pair vs two pair (tie
    )";
214         int expected = 0;
215         int actual = hand.compareTo(other);
216 
217         Testing.assertEquals(msg, expected, actual);
218     }
219 
220     private void pairVsHighCard() {
221         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
222                 new Card(14, "Diamonds"), new Card(8, "Hearts
    "), new Card(9, "Hearts"),
223                 new Card(10, "Hearts"))));
224         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
```

```
225                 new Card(13, "Diamonds"), new Card(8, "Hearts
    "), new Card(9, "Hearts"),
226                 new Card(10, "Hearts"))));
227         String msg = "Start testing pair vs high card";
228         int expected = 1;
229         int actual = hand.compareTo(other);
230 
231         Testing.assertEquals(msg, expected, actual);
232     }
233 
234     private void pairVsPairHighCard1() {
235         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
236                 new Card(14, "Diamonds"), new Card(8, "Hearts
    "), new Card(9, "Hearts"),
237                 new Card(11, "Hearts"))));
238         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(13, "Hearts"),
239                 new Card(13, "Diamonds"), new Card(8, "Hearts
    "), new Card(9, "Hearts"),
240                 new Card(10, "Hearts"))));
241         String msg = "Start testing pair vs pair (first high
    pair card)";
242         int expected = 1;
243         int actual = hand.compareTo(other);
244 
245         Testing.assertEquals(msg, expected, actual);
246     }
247 
248     private void pairVsPairHighCard2() {
249         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
250                 new Card(14, "Diamonds"), new Card(8, "Hearts
    "), new Card(9, "Hearts"),
251                 new Card(11, "Hearts"))));
252         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
253                 new Card(14, "Diamonds"), new Card(8, "Hearts
    "), new Card(9, "Hearts"),
254                 new Card(10, "Hearts"))));
255         String msg = "Start testing pair vs pair (second pair
```

```
255  card)";
256         int expected = 1;
257         int actual = hand.compareTo(other);
258 
259         Testing.assertEquals(msg, expected, actual);
260     }
261 
262     private void pairVsPairHighCard3() {
263         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
264                 new Card(14, "Diamonds"), new Card(8, "Hearts
    "), new Card(10, "Hearts"),
265                 new Card(11, "Hearts"))));
266         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
267                 new Card(14, "Diamonds"), new Card(8, "Hearts
    "), new Card(9, "Hearts"),
268                 new Card(11, "Hearts"))));
269         String msg = "Start testing pair vs pair (third pair
    card)";
270         int expected = 1;
271         int actual = hand.compareTo(other);
272 
273         Testing.assertEquals(msg, expected, actual);
274     }
275 
276     private void pairVsPairHighCard4() {
277         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
278                 new Card(14, "Diamonds"), new Card(9, "Hearts
    "), new Card(10, "Hearts"),
279                 new Card(11, "Hearts"))));
280         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
281                 new Card(14, "Diamonds"), new Card(8, "Hearts
    "), new Card(10, "Hearts"),
282                 new Card(11, "Hearts"))));
283         String msg = "Start testing pair vs pair (fourth pair
     card)";
284         int expected = 1;
285         int actual = hand.compareTo(other);
```

```
286
287         Testing.assertEquals(msg, expected, actual);
288     }
289
290     private void pairVsPairTie() {
291         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
292                 new Card(14, "Diamonds"), new Card(9, "Hearts
    "), new Card(10, "Hearts"),
293                 new Card(11, "Hearts"))));
294         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
295                 new Card(14, "Diamonds"), new Card(9, "Hearts
    "), new Card(10, "Hearts"),
296                 new Card(11, "Hearts"))));
297         String msg = "Start testing pair vs pair (tie)";
298         int expected = 0;
299         int actual = hand.compareTo(other);
300
301         Testing.assertEquals(msg, expected, actual);
302     }
303
304     private void highCardVsHighCard1() {
305         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
306                 new Card(13, "Diamonds"), new Card(12, "
    Hearts"), new Card(11, "Hearts"),
307                 new Card(10, "Hearts"))));
308         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(13, "Hearts"),
309                 new Card(12, "Diamonds"), new Card(11, "
    Hearts"), new Card(10, "Hearts"),
310                 new Card(9, "Hearts"))));
311         String msg = "Start testing high card vs high card (
    first high card)";
312         int expected = 1;
313         int actual = hand.compareTo(other);
314
315         Testing.assertEquals(msg, expected, actual);
316     }
317
```

```
318     private void highCardVsHighCard2() {
319         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
320                 new Card(13, "Diamonds"), new Card(12, "
    Hearts"), new Card(11, "Hearts"),
321                 new Card(10, "Hearts"))));
322         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
323                 new Card(12, "Diamonds"), new Card(11, "
    Hearts"), new Card(10, "Hearts"),
324                 new Card(9, "Hearts"))));
325         String msg = "Start testing high card vs high card (
    second high card)";
326         int expected = 1;
327         int actual = hand.compareTo(other);
328 
329         Testing.assertEquals(msg, expected, actual);
330     }
331 
332     private void highCardVsHighCard3() {
333         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
334                 new Card(13, "Diamonds"), new Card(12, "
    Hearts"), new Card(11, "Hearts"),
335                 new Card(10, "Hearts"))));
336         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
337                 new Card(13, "Diamonds"), new Card(11, "
    Hearts"), new Card(10, "Hearts"),
338                 new Card(9, "Hearts"))));
339         String msg = "Start testing high card vs high card (
    third high card)";
340         int expected = 1;
341         int actual = hand.compareTo(other);
342 
343         Testing.assertEquals(msg, expected, actual);
344     }
345 
346     private void highCardVsHighCard4() {
347         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
```

```
348                 new Card(13, "Diamonds"), new Card(12, "
    Hearts"), new Card(11, "Hearts"),
349                 new Card(10, "Hearts"))));
350         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
351                 new Card(13, "Diamonds"), new Card(12, "
    Hearts"), new Card(10, "Hearts"),
352                 new Card(9, "Hearts"))));
353         String msg = "Start testing high card vs high card (
    fourth high card)";
354         int expected = 1;
355         int actual = hand.compareTo(other);
356 
357         Testing.assertEquals(msg, expected, actual);
358     }
359 
360     private void highCardVsHighCard5() {
361         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
362                 new Card(13, "Diamonds"), new Card(12, "
    Hearts"), new Card(11, "Hearts"),
363                 new Card(10, "Hearts"))));
364         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
365                 new Card(13, "Diamonds"), new Card(12, "
    Hearts"), new Card(11, "Hearts"),
366                 new Card(9, "Hearts"))));
367         String msg = "Start testing high card vs high card (
    fifth high card)";
368         int expected = 1;
369         int actual = hand.compareTo(other);
370 
371         Testing.assertEquals(msg, expected, actual);
372     }
373 
374     private void highCardVsHighCardTie() {
375         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
376                 new Card(13, "Diamonds"), new Card(12, "
    Hearts"), new Card(11, "Hearts"),
377                 new Card(10, "Hearts"))));
```

```
378         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, "Hearts"),
379                 new Card(13, "Diamonds"), new Card(12, "
    Hearts"), new Card(11, "Hearts"),
380                 new Card(10, "Hearts"))));
381         String msg = "Start testing high card vs high card (
    tie)";
382         int expected = 0;
383         int actual = hand.compareTo(other);
384 
385         Testing.assertEquals(msg, expected, actual);
386     }
387 }
```