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

package proj4;

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

public class Card {

    private final String[] STRING_RANKS = new String[] {"two", "three", "four", "five", "six", "seven", "eight", "nine",
            "ten", "jack", "queen", "king", "ace"};
    private final String[] STRING_SUITS = new String[] {"Spades", "Hearts", "Clubs", "Diamonds"};
    private final int DEFAULT_RANK = 14;
    private final String DEFAULT_SUIT = "Spades";
    private final String[] STRING_NUMERIC_RANKS = new String[] {"2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12",
            "13", "14"};
    private final int[] INT_RANKS = new int[] {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14};
    private final int[] INT_SUITS = new int[] {0, 1, 2, 3};

    private int rank;
    private String suit;


    /**
     * Constructor
     * @param stringRank String: whole cards (2-10) can either be spelled
     * out like "two" or numeric like "2". Face cards will always be
     * spelled out like "Jack". Case insensitive.
     * @param stringSuit String: "Spades", "Hearts", "Clubs", or "Diamonds"
     */
    public Card(String stringRank, String stringSuit) {
        stringRank = stringRank.toLowerCase();
```

```
35          stringSuit = getStandard(stringSuit);
36
37          if (isIn(stringRank, STRING_RANKS) && isIn(stringSuit
    , STRING_SUITS)) {
38              rank = toNumericRank(stringRank);
39              suit = stringSuit;
40          }
41
42          else if (isIn(stringRank, STRING_NUMERIC_RANKS) &&
    isIn(stringSuit, STRING_SUITS)) {
43              rank = Integer.valueOf(stringRank);
44              suit = stringSuit;
45          }
46
47          else {
48              rank = DEFAULT_RANK;
49              suit = DEFAULT_SUIT;
50
51          }
52      }
53
54      /**
55       * Constructor
56       * @param intRank integer between 2-14
57       * @param intSuit integer: 0=Spades, 1=Hearts, 2=Clubs, or
     3=Diamonds
58       */
59      public Card(int intRank, int intSuit) {
60          if (isIn(intRank, INT_RANKS) && isIn(intSuit,
    INT_SUITS)) {
61              rank = intRank;
62              suit = STRING_SUITS[intSuit];
63          }
64
65          else {
66              rank = DEFAULT_RANK;
67              suit = DEFAULT_SUIT;
68          }
69      }
70
71      /**
```

```
72      * Getter for the rank
73      * @retrun an integer for the rank
74      */
75     public int getRank() {
76         return rank;
77     }
78 
79     /**
80      * Getter for the suit
81      * @retrun an integer for the suit
82      */
83     public String getSuit() {
84         return suit;
85     }
86 
87     /**
88      * Returns the readable version of the card
89      * @return a string for the readable version of the card
90      */
91     public String toString() {
92         return getStringRank() + " of " + getSuit();
93     }
94 
95     /**
96      * Gets the standard version of the given string
97      * @param myString a string for the non-standard version
98      * @return a string for the standard version
99      */
100    private String getStandard(String myString) {
101        return myString.substring(0, 1).toUpperCase() +
    myString.substring(1).toLowerCase();
102    }
103 
104    /**
105     * Checks whether the value is in the array
106     * @param value a string to check
107     * @param array a string array to check
108     * @return true if the value is in the array
109     */
110    private boolean isIn(String value, String[] array) {
111        return Arrays.asList(array).contains(value);
```

```
112     }
113
114     /**
115      * Gets the numeric version of the rank
116      * @param stringRank a string for the rank
117      * @return a string for the numeric version of the rank
118      */
119     private int toNumericRank(String stringRank) {
120         return INT_RANKS[Arrays.asList(STRING_RANKS).indexOf(
    stringRank)];
121     }
122
123     /**
124      * Checks whether the value is in the array
125      * @param value an integer to check
126      * @param array an integer array to check
127      * @return true if the value is in the array
128      */
129     private boolean isIn(int value, int[] array) {
130         ArrayList<Integer> intArrayList = new ArrayList<
    Integer>();
131
132         for (int i : array) {
133             intArrayList.add(i);
134         }
135
136         return intArrayList.contains(value);
137     }
138
139         /**
140          * Gets the string version of the rank
141          * @return a string for the string version of the
    rank
142          */
143         private String getStringRank() {
144             int currentRank = getRank();
145             if (currentRank > 10) {
146                 return getStandard(STRING_RANKS[currentRank
     - 2]);
147             }
148
```

```
149                 else {
150                     return currentRank + "";
151                 }
152             }
153 }
```

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

package proj4;

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 int[] SUITS = new int[] {0, 1, 2, 3};
    private final int MAX_DECK = 52;
    private final int EMPTY = 0;

    private ArrayList<Card> deck;
    private int nextToDeal;

    /**
     * Default constructor for the deck
     */
    public Deck() {
        deck = new ArrayList<Card>();

        for (int rank : RANKS) {
            for (int suit : SUITS) {
                deck.add(new Card(rank, suit));
            }
        }

        nextToDeal = START;
    }

    /**
     * Shuffles the deck
     */
    public void shuffle() {
        for (int i = getNextToDeal(); i < MAX_DECK; i++) {
```

```
41              int randomNumber = ThreadLocalRandom.current().
   nextInt(getNextToDeal(), MAX_DECK);
42              Card currentCard = getDeck().get(i);
43              Card randomCard = getDeck().get(randomNumber);
44
45              getDeck().set(i, randomCard);
46              getDeck().set(randomNumber, currentCard);
47          }
48      }
49
50      /**
51       * Returns the next undealt card or null if deck is empty
52       * @return the next undealt card or null if deck is empty
53       */
54      public Card deal() {
55          if (getDeck().isEmpty()) {
56              return null;
57          }
58
59          else {
60              Card currentCard = getDeck().get(getNextToDeal());
61
62              nextToDeal++;
63
64              return currentCard;
65          }
66      }
67
68      /**
69       * Resets the deck to a state where all cards are undealt
   in a shuffled state
70       */
71      public void gather() {nextToDeal = START;}
72
73      /**
74       * Returns the number of undealt cards in the deck
75       * @return the number of undealt cards in the deck
76       */
77      public int size() {return MAX_DECK - getNextToDeal();}
78
79      /**
```

```
80       * Checks whether the deck is empty
81       * @return true if there are still undealt cards in the
   deck
82       */
83      public boolean isEmpty() {
84          if (size() == EMPTY){
85              return true;
86          }
87
88          else {
89              return false;
90          }
91      }
92
93      /**
94       * Returns all the undealt cards in the deck as a string
95       * @return all the undealt cards in the deck as a string
96       */
97      public String toString() {
98          String returnString = "";
99
100         for (int i = getNextToDeal(); i < MAX_DECK; i++) {
101             returnString += getDeck().get(i) + "\n";
102         }
103
104         return returnString;
105     }
106
107     /**
108      * Gets the deck
109      * @return a Deck object for the deck
110      */
111     private ArrayList<Card> getDeck() {
112         return deck;
113     }
114
115     /**
116      * Gets the value of nextToDeal
117      * @return an integer for the nextToDeal
118      */
119     private int getNextToDeal() {
```

```
120         return nextToDeal;
121     }
122 
123     /**
124      * Deals the card i times
125      * @return an array list for the card list
126      */
127     public ArrayList dealITimes(int i){
128         ArrayList<Card> cardList = new ArrayList<Card>();
129 
130         for (int j = START; j < i; j++) {
131             cardList.add(deal());
132         }
133 
134         return cardList;
135     }
136 }
```

```java
/**
 * A simple texas hold'em poker game
 */

package proj4;

import java.util.Scanner;
import java.util.Arrays;

public class Client {

    private final boolean CONTINUE = true;
    private final int MAX_COLLECTION = 5;
    private final int MAX_HAND = 2;
    private final String[] VALID_ANSWERS = new String[] {"my hand", "other hand", "tie"};

    /**
     * @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.
     */
    public static void main(String[] args) {
        Client client = new Client();
        Deck deck = new Deck();
        deck.shuffle();
        CommunityCardSet communityCardSet = new CommunityCardSet(deck.dealITimes(client.MAX_COLLECTION));
        boolean continueGame = client.CONTINUE;
        int totalPoint = 0;

        while (continueGame && deck.size() ⩾ client.MAX_HAND * 2) {
            StudPokerHand myHand = new StudPokerHand(communityCardSet, deck.dealITimes(client.MAX_HAND));
            StudPokerHand otherHand = new StudPokerHand(communityCardSet, deck.dealITimes(client.MAX_HAND));
            String result = myHand.getResult(otherHand);

```

```
35              client.printInstruction(communityCardSet, myHand,
   otherHand);
36
37              if (client.getValidAnswer().equals(result)) {
38                  System.out.println("CORRECT\n
   ------------------------------------------------");
39                  totalPoint++;
40              }
41
42              else {
43                  continueGame = ! client.CONTINUE;
44              }
45          }
46
47          System.out.println("Game is over, and your total point
    is " + totalPoint);
48      }
49
50      /**
51       * Prints the instruction
52       * @param communityCardSet CommunityCardSet object for the
    community card set
53       * @param myHand StudPokerHand object for the hand
54       * @param otherHand StudPokerHand object for another hand
55       */
56      private void printInstruction(CommunityCardSet
   communityCardSet, StudPokerHand myHand, StudPokerHand
   otherHand) {
57          System.out.println("community card set: " +
   communityCardSet);
58          System.out.println("Who is the winner?");
59          System.out.println("my hand: " + myHand);
60          System.out.println("other hand: " + otherHand);
61      }
62
63      /**
64       * Gets the valid answer from the user
65       * @return a string for the valid answer
66       */
67      private String getValidAnswer() {
68          Scanner sc = new Scanner(System.in);
```

```
69          System.out.println("Type my hand, other hand, or tie"
    );
70
71          String userInput = sc.nextLine();
72
73          while (! Arrays.asList(VALID_ANSWERS).contains(
    userInput)) {
74              System.out.println("Your answer is invalid. Type
    my hand, other hand, or tie");
75              userInput = sc.nextLine();
76          }
77          return userInput;
78      }
79  }
```

```java
package proj4;

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

    private static boolean VERBOSE = 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 poker hand
 */

package proj4;

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> pokerHand;

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

    /**
     * Adds the card to the hand if the hand does not have 5 cards in it
     * @param card a proj4.Card object that will be added to the hand
     */
    public void addCard(Card card) {
        if (getPokerHand().size() < MAX_HAND) {
            getPokerHand().add(card);
```

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

```
76       * @return a negtive number if this is worth LESS than
   other, zero
77       * if they are worth the SAME, and a positive number if
   this is worth
78       * MORE than other
79       */
80      public int compareTo(PokerHand other) {
81          int myPoint = getPoint();
82          int otherPoint = other.getPoint();
83
84          int result = myPoint - otherPoint;
85
86          if (result == TIE) {
87              if (myPoint == FLUSH || myPoint == HIGH_CARD) {
88                  return compareFlushHighCard(other);
89              }
90
91              else if (myPoint == TWO_PAIR || myPoint == PAIR
   ) {
92                  return compareTwoPairPair(other);
93              }
94
95              else {
96                  System.out.println("These hands are invalid"
   );
97              }
98          }
99
100         return result;
101     }
102
103     /**
104      * Gets the poker hand
105      * @return an array list for the poker hand
106      */
107     private ArrayList<Card> getPokerHand() {
108         return pokerHand;
109     }
110
111     /** Gets the point of the hand
112      * @return an integer for the point of the hand
```

```
113      */
114     private int getPoint() {
115         int handScore;
116 
117         if (isFlush()){
118             handScore = FLUSH;
119         }
120 
121         else if (isTwoPair()){
122             handScore = TWO_PAIR;
123         }
124 
125         else if (isPair()){
126             handScore = PAIR;
127         }
128 
129         else {
130             handScore = HIGH_CARD;
131         }
132 
133         return handScore;
134     }
135 
136     /**
137      * Compares two flush or high card hands
138      * @param other a PokerHand object that is compared with
    this hand
139      * @return an integer for the result
140      */
141     private int compareFlushHighCard (PokerHand other) {
142         ArrayList<Integer> myFlushRankList = getFlushRankList
    ();
143         ArrayList<Integer> otherFlushRankList = other.
    getFlushRankList();
144 
145         for (int i = START; i < MAX_HAND; i++) {
146             int result = getIthRank(i, myFlushRankList) -
    getIthRank(i, otherFlushRankList);
147 
148             if (result != TIE) {
149                 return result;
```

```
150             }
151         }
152 
153         return TIE;
154     }
155 
156     /**
157      * Compares two two pair or pair hands
158      * @param other a PokerHand object that is compared with
    this hand
159      * @return an integer for the result
160      */
161     private int compareTwoPairPair(PokerHand other) {
162         ArrayList<Integer> myPairRankList = getPairRankList
    ();
163         ArrayList<Integer> otherPairRankList = other.
    getPairRankList();
164 
165         for (int i = START; i < myPairRankList.size(); i++) {
166             int result = getIthRank(i, myPairRankList) -
    getIthRank(i, otherPairRankList);
167 
168             if (result != TIE) {
169                 return result;
170             }
171         }
172 
173         return TIE;
174     }
175 
176     /**
177      * Gets the flush rank list
178      * @return an array list for the flush rank list
179      */
180     private ArrayList<Integer> getFlushRankList() {
181         ArrayList<Integer> rankFlushList = new ArrayList<
    Integer>();
182 
183         for (Card card : getPokerHand()) {
184             rankFlushList.add(card.getRank());
185         }
```

```
186
187         rankFlushList.sort(Comparator.reverseOrder());
188
189         return rankFlushList;
190     }
191
192     /**
193      * Gets the pair rank list
194      * @return an array list for the pair rank list
195      */
196     private ArrayList<Integer> getPairRankList() {
197         PokerHand currentHand = new PokerHand(getPokerHand
    ());
198
199         int i = 1;
200         ArrayList<Integer> pairRankList = new ArrayList<
    Integer>();
201         ArrayList<Integer> otherRankList = new ArrayList<
    Integer>();
202
203         while (currentHand.getPokerHand().size() > i) {
204             if (currentHand.getIthCard(FIRST_CARD).getRank
    () == currentHand.getIthCard(i).getRank()) {
205                 pairRankList.add(currentHand.getIthCard(
    FIRST_CARD).getRank());
206
207                 currentHand.removeIthCard(i);
208                 currentHand.removeIthCard(FIRST_CARD);
209
210                 i = 1;
211             } else {
212                 i++;
213             }
214
215             if (currentHand.getPokerHand().size() == i) {
216                 otherRankList.add(currentHand.getIthCard(
    FIRST_CARD).getRank());
217
218                 currentHand.removeIthCard(FIRST_CARD);
219
220                 i = 1;
```

```
221             }
222
223             if (currentHand.getPokerHand().size() ==
    LAST_CARD) {
224                 otherRankList.add(currentHand.getIthCard(
    FIRST_CARD).getRank());
225
226                 currentHand.removeIthCard(FIRST_CARD);
227             }
228         }
229
230         pairRankList.sort(Comparator.reverseOrder());
231         otherRankList.sort(Comparator.reverseOrder());
232
233         pairRankList.addAll(otherRankList);
234
235         return pairRankList;
236     }
237
238     /**
239      * Getter for the rank at the given index
240      * @param index an integer for the index of the rank
241      * @rank an integer at the given index
242      */
243     private int getIthRank(int index, ArrayList<Integer>
    rankList) {return rankList.get(index);}
244
245     /**
246      * Removes the card with the given index
247      * @param index an integer for the index of the card
248      */
249     private void removeIthCard(int index) {
250         if (index < getPokerHand().size()) {
251             getPokerHand().remove(index);
252         }
253     }
254
255     /**
256      * Checks whether the hand is flush
257      * @return true if all cards have the same suit
258      */
```

```
259     private boolean isFlush() {
260         for (int i = 1; i < MAX_HAND; i++) {
261             if (getIthCard(i).getSuit() ≠ getIthCard(i - 1).
    getSuit()) {
262                 return false;
263             }
264         }
265 
266         return true;
267     }
268 
269     /**
270      * Checks whether the hand is two pair
271      * @return true if the hand has 2 pairs of the same rank
272      */
273     private boolean isTwoPair() {
274         PokerHand currentHand = new PokerHand(getPokerHand
    ());
275 
276         int i = 1;
277         int totalPair = 0;
278 
279         while (currentHand.getPokerHand().size() > i) {
280             if (currentHand.getIthCard(FIRST_CARD).getRank
    () == currentHand.getIthCard(i).getRank()) {
281                 totalPair++;
282 
283                 currentHand.removeIthCard(i);
284                 currentHand.removeIthCard(FIRST_CARD);
285 
286                 i = 1;
287             }
288 
289             else {
290                 i++;
291             }
292 
293             if (currentHand.getPokerHand().size() == i) {
294                 currentHand.removeIthCard(FIRST_CARD);
295 
296                 i = 1;
```

```
297             }
298 
299             if (currentHand.getPokerHand().size() ==
    LAST_CARD) {
300                 currentHand.removeIthCard(FIRST_CARD);
301             }
302         }
303         if (totalPair == 2) {
304             return true;
305         }
306 
307         return false;
308     }
309     /**
310      * Checks whether the hand is a pair
311      * @return true if the hand has a pair of the same rank
312      */
313     private boolean isPair() {
314         PokerHand currentHand = new PokerHand(getPokerHand
    ());
315 
316         int i = 1;
317         int totalPair = 0;
318 
319         while (currentHand.getPokerHand().size() > i) {
320             if (currentHand.getIthCard(FIRST_CARD).getRank
    () == currentHand.getIthCard(i).getRank()) {
321                 totalPair++;
322 
323                 currentHand.removeIthCard(i);
324                 currentHand.removeIthCard(FIRST_CARD);
325 
326                 i = 1;
327             }
328 
329             else {
330                 i++;
331             }
332 
333             if (currentHand.getPokerHand().size() == i) {
334                 currentHand.removeIthCard(FIRST_CARD);
```

```
335
336                 i = 1;
337             }
338
339             if (currentHand.getPokerHand().size() ==
    LAST_CARD) {
340                 currentHand.removeIthCard(FIRST_CARD);
341             }
342         }
343
344         if (totalPair == 1) {
345             return true;
346         }
347
348         return false;
349     }
350 }
```

```java
package proj4;

public class CardTester {

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

        Testing.startTests();
        cardTester.testStringCardConstructor();
        cardTester.testIntCardConstructor();
        cardTester.testGetRank();
        cardTester.testGetSuit();
        Testing.finishTests();
    }

    private void testStringCardConstructor() {
        stringCardConstructor1();
        stringCardConstructor2();
        stringCardConstructor3();
        invalidStringCardConstructor1();
        invalidStringCardConstructor2();
    }
    private void stringCardConstructor1() {
        Card card = new Card("Two", "clubs");
        String msg = "Starts testing string card constructor1 and toString method";
        String expected = "2 of Clubs";
        String actual = card.toString();

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

    private void stringCardConstructor2() {
        Card card = new Card("2", "clubs");
        String msg = "Starts testing string card constructor2 and toString method";
        String expected = "2 of Clubs";
        String actual = card.toString();

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

```
40
41      private void stringCardConstructor3() {
42          Card card = new Card("jack", "clubs");
43          String msg = "Starts testing string card constructor3
    and toString method";
44          String expected = "Jack of Clubs";
45          String actual = card.toString();
46
47          Testing.assertEquals(msg, expected, actual);
48      }
49
50      private void invalidStringCardConstructor1() {
51          Card card = new Card("Invalid String", "clubs");
52          String msg = "Starts testing invalid string card
    constructor1 and toString method";
53          String expected = "Ace of Spades";
54          String actual = card.toString();
55
56          Testing.assertEquals(msg, expected, actual);
57      }
58
59      private void invalidStringCardConstructor2() {
60          Card card = new Card("Two", "Invalid String");
61          String msg = "Starts testing invalid string card
    constructor2 and toString method";
62          String expected = "Ace of Spades";
63          String actual = card.toString();
64
65          Testing.assertEquals(msg, expected, actual);
66      }
67
68      private void testIntCardConstructor() {
69          intCardConstructor();
70          invalidIntCardConstructor1();
71          invalidIntCardConstructor2();
72      }
73
74      private void intCardConstructor() {
75          Card card = new Card(2, 2);
76          String msg = "Starts testing int card constructor and
    toString method";
```

```
 77          String expected = "2 of Clubs";
 78          String actual = card.toString();
 79
 80          Testing.assertEquals(msg, expected, actual);
 81      }
 82
 83      private void invalidIntCardConstructor1() {
 84          Card card = new Card(15, 2);
 85          String msg = "Starts testing invalid int card
    constructor1 and toString method";
 86          String expected = "Ace of Spades";
 87          String actual = card.toString();
 88
 89          Testing.assertEquals(msg, expected, actual);
 90      }
 91
 92      private void invalidIntCardConstructor2() {
 93          Card card = new Card(2, 4);
 94          String msg = "Starts testing invalid int card
    constructor2 and toString method";
 95          String expected = "Ace of Spades";
 96          String actual = card.toString();
 97
 98          Testing.assertEquals(msg, expected, actual);
 99      }
100
101      private void testGetRank() {
102          getRank1();
103          getRank2();
104      }
105      private void getRank1() {
106          Card card = new Card(2, 2);
107          String msg = "Starts testing getRank1 method";
108          int expected = 2;
109          int actual = card.getRank();
110
111          Testing.assertEquals(msg, expected, actual);
112      }
113
114      private void getRank2() {
115          Card card = new Card("two", "Clubs");
```

```
116         String msg = "Starts testing getRank2 method";
117         int expected = 2;
118         int actual = card.getRank();
119 
120         Testing.assertEquals(msg, expected, actual);
121     }
122 
123     private void testGetSuit() {
124        getSuit1();
125        getSuit2();
126     }
127 
128     private void getSuit1() {
129         Card card = new Card(2, 2);
130         String msg = "Starts testing getSuit1 method";
131         String expected = "Clubs";
132         String actual = card.getSuit();
133 
134         Testing.assertEquals(msg, expected, actual);
135     }
136 
137     private void getSuit2() {
138         Card card = new Card("two", "Clubs");
139         String msg = "Starts testing getSuit2 method";
140         String expected = "Clubs";
141         String actual = card.getSuit();
142 
143         Testing.assertEquals(msg, expected, actual);
144     }
145 }
```

```java
package proj4;

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

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

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

```
65
66      private void testDeal() {
67          Deck deck = new Deck();
68          String msg = "Starts testing deal method";
69          Card expected = new Card(2, 0);
70          Card actual = deck.deal();
71
72          Testing.assertEquals(msg, expected.toString(), actual
    .toString());
73      }
74
75      private void testGather() {
76          Deck deck = new Deck();
77
78          deck.dealITimes(MAX_DECK);
79          deck.gather();
80
81          String msg = "Starts testing gather method";
82          String expected = "2 of Spades\n" + "2 of Hearts\n"
     + "2 of Clubs\n" + "2 of Diamonds\n" + "3 of Spades\n"
83                  + "3 of Hearts\n" + "3 of Clubs\n" + "3 of
    Diamonds\n" + "4 of Spades\n" + "4 of Hearts\n" + "4 of Clubs
    \n"
84                  + "4 of Diamonds\n" + "5 of Spades\n" + "5 of
     Hearts\n" + "5 of Clubs\n" + "5 of Diamonds\n" + "6 of
    Spades\n"
85                  + "6 of Hearts\n" + "6 of Clubs\n" + "6 of
    Diamonds\n" + "7 of Spades\n" + "7 of Hearts\n" + "7 of Clubs
    \n"
86                  + "7 of Diamonds\n" + "8 of Spades\n" + "8 of
     Hearts\n" + "8 of Clubs\n" + "8 of Diamonds\n" + "9 of
    Spades\n"
87                  + "9 of Hearts\n" + "9 of Clubs\n" + "9 of
    Diamonds\n" + "10 of Spades\n" + "10 of Hearts\n" + "10 of
    Clubs\n"
88                  + "10 of Diamonds\n" + "Jack of Spades\n" + "
    Jack of Hearts\n" + "Jack of Clubs\n" + "Jack of Diamonds\n"
89                  + "Queen of Spades\n" + "Queen of Hearts\n"
     + "Queen of Clubs\n" + "Queen of Diamonds\n" + "King of
    Spades\n"
90                  + "King of Hearts\n" + "King of Clubs\n" + "
```

```
 90 King of Diamonds\n" + "Ace of Spades\n" + "Ace of Hearts\n"
 91                 + "Ace of Clubs\n" + "Ace of Diamonds\n";
 92         Deck actual = deck;
 93 
 94         Testing.assertEquals(msg, expected, actual.toString
    ());
 95     }
 96 
 97     private void testIsEmpty() {
 98         testNotEmptyDeck();
 99         testEmptyDeck();
100     }
101 
102     private void testNotEmptyDeck() {
103         Deck deck = new Deck();
104         String msg = "Starts testing not empty deck";
105         boolean expected = false;
106         boolean actual = deck.isEmpty();
107 
108         Testing.assertEquals(msg, expected, actual);
109     }
110 
111     private void testEmptyDeck() {
112         Deck deck = new Deck();
113 
114         deck.dealITimes(MAX_DECK);
115 
116         String msg = "Starts testing empty deck";
117         boolean expected = true;
118         boolean actual = deck.isEmpty();
119 
120         Testing.assertEquals(msg, expected, actual);
121     }
122 
123     private void testSize() {
124         Deck deck = new Deck();
125         String msg = "Starts testing size method";
126         int expected = MAX_DECK;
127 
128         for (int i = START; i < MAX_DECK; i++) {
129             deck.deal();
```

```
130             expected--;
131 
132             int actual = deck.size();
133 
134             Testing.assertEquals(msg, expected, actual);
135         }
136     }
137 
138     private void testDealITimes() {
139         Deck deck = new Deck();
140         String msg = "Starts testing dealITimes method";
141         ArrayList<Card> expected = new ArrayList<Card>();
142 
143         expected.add(new Card(2, 0));
144         expected.add(new Card(2, 1));
145         expected.add(new Card(2, 2));
146         expected.add(new Card(2, 3));
147         expected.add(new Card(3, 0));
148 
149         ArrayList<Card> actual = deck.dealITimes(5);
150 
151         Testing.assertEquals(msg, expected.toString(), actual
    .toString());
152     }
153 }
```

```java
/**
 * Models a 2-card poker hand
 */

package proj4;

import java.util.ArrayList;

public class StudPokerHand {

    private final int MAX_HAND = 2;
    private final int FIRST_CARD = 0;
    private final int START = 0;
    private final int MAX_CANDIDATE = 7;

    private CommunityCardSet collection;
    private ArrayList<Card> twoCardHand;

    /**
     * Non-default constructor for the stud poker hand
     * @param cc a CommunityCardSet object for the community card set
     * @param cardList an array list for the hand
     */
    public StudPokerHand(CommunityCardSet cc, ArrayList<Card> cardList) {
        collection = new CommunityCardSet(cc.getCollection());
        twoCardHand = new ArrayList<Card>(cardList);
    }

    /**
     * Adds the card to the hand if the hand does not have 2 cards
     * @param card a proj4.Card object that will be added to the hand
     */
    public void addCard(Card card) {
        if (getTwoCardHand().size() < MAX_HAND) {
            getTwoCardHand().add(card);
        }
    }
```

```
38
39      /**
40       * Getter for the card at the given index
41       * @param index an integer greater or equal to 0
42       * @return a card object at the given index or null if
   index is invalid
43       */
44      public Card getIthCard(int index){
45          if (FIRST_CARD ⩽ index && index < getTwoCardHand().
   size()) {
46              return getTwoCardHand().get(index);
47          }
48
49          else {
50              return null;
51          }
52      }
53
54      /**
55       * Returns the readable version of the hand
56       * @return a string for the readable version of the hand
57       */
58      public String toString(){
59          String returnString = "";
60
61          for (Card card : getTwoCardHand()) {
62              returnString += card + "\n";
63          }
64
65          return returnString;
66      }
67
68      /**
69       * Determines how this hand compares to another hand,
   using the
70       * community card set to determine the best 5-card hand it
    can
71       * make. Returns positive, negative, or zero depending on
   the comparison.
72       *
73       * @param other The hand to compare this hand to
```

```
74       * @return a negative number if this is worth LESS than
   other, zero
75       * if they are worth the SAME, and a positive number if
   this is worth
76       * MORE than other
77       */
78      public int compareTo(StudPokerHand other) {
79          return getBestFiveCardHand().compareTo(other.
   getBestFiveCardHand());
80      }
81
82      /**
83       * Gets the best poker hand among all poker hands
84       * @return an array list for the best poker hand
85       */
86      private PokerHand getBestFiveCardHand() {
87          ArrayList<PokerHand> hands = getAllFiveCardHands();
88          PokerHand bestSoFar = hands.get(START);
89
90          for (int i = 1; i < hands.size(); i++) {
91              if (hands.get(i).compareTo(bestSoFar) > 0) {
92                  bestSoFar = hands.get(i);
93              }
94          }
95
96          return bestSoFar;
97      }
98
99      /**
100      * Gets all possible poker hands
101      * @return an array list of poker hands
102      */
103     private ArrayList<PokerHand> getAllFiveCardHands() {
104         ArrayList<Card> candidate = getCandidate();
105         ArrayList<PokerHand> allFiveCardHands = new ArrayList
   <PokerHand>();
106
107         for (int i = START; i < MAX_CANDIDATE; i++) {
108             for (int j = i + 1; j < MAX_CANDIDATE; j++) {
109                 candidate.remove(j);
110                 candidate.remove(i);
```

```
111
112                 allFiveCardHands.add(new PokerHand(candidate
    ));
113
114                 candidate = getCandidate();
115             }
116         }
117
118         return allFiveCardHands;
119     }
120
121     /**
122      * Gets the candidate for the poker hand
123      * @return an array list for the candidate
124      */
125     private ArrayList<Card> getCandidate() {
126         ArrayList<Card> candidate = new ArrayList<Card>();
127         candidate.addAll(getCollection());
128         candidate.addAll(getTwoCardHand());
129
130         return candidate;
131     }
132
133     /**
134      * Gets the collection of the community card set
135      * @return an array list for the collection
136      */
137     private ArrayList<Card> getCollection() {
138         return collection.getCollection();
139     }
140
141     /**
142      * Gets two card hand
143      * @return an array list for the two card hand
144      */
145     private ArrayList<Card> getTwoCardHand() {
146         return twoCardHand;
147     }
148
149     /**
150      * Gets the result according to the given value
```

```java
     * @return a string according to the given value
     */
    public String getResult(StudPokerHand other) {
        int result = compareTo(other);

        if (result > 0) {
            return "my hand";
        }

        else if (result < 0) {
            return "other hand";
        }

        else {
            return "tie";
        }
    }
}
```

```java
package proj4;

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

public class PokerHandTester {

    private final int MAX_HAND = 5;

    public static void main(String[] args) {
        PokerHandTester pokerHandTester = new PokerHandTester
();

        Testing.startTests();
        pokerHandTester.testPokerHandConstructor();
        pokerHandTester.testAddCard();
        pokerHandTester.testGetIthCard();
        pokerHandTester.testCompareTo();
        Testing.finishTests();
    }

    private void testPokerHandConstructor() {
        Deck deck = new Deck();
        PokerHand hand = new PokerHand(deck.dealITimes(
MAX_HAND));
        String msg = "Start testing PokerHand constructor and
toString method";
        String expected = "2 of Spades\n" + "2 of Hearts\n" +
"2 of Clubs\n" + "2 of Diamonds\n" + "3 of Spades\n";
        PokerHand actual = hand;

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

    private void testAddCard() {
        testNotFullHand();
        testFullHand();
    }

    private void testNotFullHand() {
        Deck deck = new Deck();
```

```
38          PokerHand hand = new PokerHand(deck.dealITimes(
    MAX_HAND - 1));
39          String msg = "Start testing addCard method in the not
    full hand";
40          String expected = "2 of Spades\n" + "2 of Hearts\n" +
    "2 of Clubs\n" + "2 of Diamonds\n" + "3 of Spades\n";
41          hand.addCard(deck.deal());
42          PokerHand actual = hand;
43
44          Testing.assertEquals(msg,expected, actual.toString());
45      }
46
47      private void testFullHand() {
48          Deck deck = new Deck();
49          PokerHand hand = new PokerHand(deck.dealITimes(
    MAX_HAND));
50          String msg = "Start testing addCard method in the full
     hand";
51          String expected = "2 of Spades\n" + "2 of Hearts\n" +
    "2 of Clubs\n" + "2 of Diamonds\n" + "3 of Spades\n";
52          hand.addCard(deck.deal());
53          PokerHand actual = hand;
54
55          Testing.assertEquals(msg,expected, actual.toString());
56      }
57
58      private void testGetIthCard() {
59          Deck deck = new Deck();
60          PokerHand hand = new PokerHand(deck.dealITimes(
    MAX_HAND));
61          String msg = "Start testing getIthCard method";
62          String expected = "2 of Hearts";
63          Card actual = hand.getIthCard(1);
64
65          Testing.assertEquals(msg,expected, actual.toString());
66      }
67
68      private void testCompareTo() {
69          flushVsTwoPair();
70          flushVsFlushHighCard1();
71          flushVsFlushHighCard2();
```

```
 72          flushVsFlushHighCard3();
 73          flushVsFlushHighCard4();
 74          flushVsFlushHighCard5();
 75          flushVsFlushTie();
 76
 77          twoPairVsPair();
 78          twoPairVsTwoPairHighCard1();
 79          twoPairVsTwoPairHighCard2();
 80          twoPairVsTwoPairHighCard3();
 81          twoPairVsTwoPairTie();
 82
 83          pairVsHighCard();
 84          pairVsPairHighCard1();
 85          pairVsPairHighCard2();
 86          pairVsPairHighCard3();
 87          pairVsPairHighCard4();
 88          pairVsPairTie();
 89
 90          highCardVsHighCard1();
 91          highCardVsHighCard2();
 92          highCardVsHighCard3();
 93          highCardVsHighCard4();
 94          highCardVsHighCard5();
 95          highCardVsHighCardTie();
 96      }
 97
 98      private void flushVsTwoPair() {
 99          PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 1),
100                  new Card(14, 1), new Card(8, 1), new Card(7,
    1),
101                  new Card(10, 1))));
102          PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 1),
103                  new Card(14, 1), new Card(8, 0), new Card(8,
    1),
104                  new Card(9, 0))));
105          String msg = "Start testing flush vs two pair";
106          int expected = 1;
107          int actual = hand.compareTo(other);
108
```

```
109         Testing.assertEquals(msg, expected, actual);
110     }
111 
112     private void flushVsFlushHighCard1() {
113         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 1),
114                 new Card(13, 1), new Card(12, 1), new Card(11
    , 1),
115                 new Card(10, 1))));
116         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(9, 1),
117                 new Card(13, 1), new Card(12, 1), new Card(11
    , 1),
118                 new Card(10, 1))));
119         String msg = "Start testing flush vs flush (first
    high card)";
120         int expected = 1;
121         int actual = hand.compareTo(other);
122 
123         Testing.assertEquals(msg, expected, actual);
124     }
125 
126     private void flushVsFlushHighCard2() {
127         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 1),
128                 new Card(13, 1), new Card(12, 1), new Card(11
    , 1),
129                 new Card(10, 1))));
130         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(9, 1),
131                 new Card(14, 1), new Card(12, 1), new Card(11
    , 1),
132                 new Card(10, 1))));
133         String msg = "Start testing flush vs flush (second
    high card)";
134         int expected = 1;
135         int actual = hand.compareTo(other);
136 
137         Testing.assertEquals(msg, expected, actual);
138     }
139 
```

```
140     private void flushVsFlushHighCard3() {
141         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 1),
142                 new Card(13, 1), new Card(12, 1), new Card(11
    , 1),
143                 new Card(10, 1))));
144         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(9, 1),
145                 new Card(14, 1), new Card(13, 1), new Card(11
    , 1),
146                 new Card(10, 1))));
147         String msg = "Start testing flush vs flush (third
    high card)";
148         int expected = 1;
149         int actual = hand.compareTo(other);
150 
151         Testing.assertEquals(msg, expected, actual);
152     }
153 
154     private void flushVsFlushHighCard4() {
155         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 1),
156                 new Card(13, 1), new Card(12, 1), new Card(11
    , 1),
157                 new Card(10, 1))));
158         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(9, 1),
159                 new Card(14, 1), new Card(13, 1), new Card(12
    , 1),
160                 new Card(10, 1))));
161         String msg = "Start testing flush vs flush (fourth
    high card)";
162         int expected = 1;
163         int actual = hand.compareTo(other);
164 
165         Testing.assertEquals(msg, expected, actual);
166     }
167 
168     private void flushVsFlushHighCard5() {
169         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 1),
```

```
170                 new Card(13, 1), new Card(12, 1), new Card(11
    , 1),
171                 new Card(10, 1))));
172         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(9, 1),
173                 new Card(14, 1), new Card(13, 1), new Card(12
    , 1),
174                 new Card(11, 1))));
175         String msg = "Start testing flush vs flush (fifth
    high card)";
176         int expected = 1;
177         int actual = hand.compareTo(other);
178 
179         Testing.assertEquals(msg, expected, actual);
180     }
181 
182     private void flushVsFlushTie() {
183         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 1),
184                 new Card(13, 1), new Card(12, 1), new Card(11
    , 1),
185                 new Card(10, 1))));
186         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(10, 1),
187                 new Card(14, 1), new Card(13, 1), new Card(12
    , 1),
188                 new Card(11, 1))));
189         String msg = "Start testing flush vs flush (tie)";
190         int expected = 0;
191         int actual = hand.compareTo(other);
192 
193         Testing.assertEquals(msg, expected, actual);
194     }
195 
196     private void twoPairVsPair() {
197         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
198                 new Card(14, 1), new Card(8, 2), new Card(8,
    0),
199                 new Card(10, 1))));
200         PokerHand other = new PokerHand(new ArrayList<Card>(
```

```
200 Arrays.asList(new Card(14, 0),
201                 new Card(14, 1), new Card(8, 2), new Card(9,
    0),
202                 new Card(10, 1))));
203         String msg = "Start testing two pair vs pair";
204         int expected = 1;
205         int actual = hand.compareTo(other);
206 
207         Testing.assertEquals(msg, expected, actual);
208     }
209 
210     private void twoPairVsTwoPairHighCard1() {
211         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
212                 new Card(14, 1), new Card(8, 2), new Card(8,
    0),
213                 new Card(9, 1))));
214         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(13, 0),
215                 new Card(13, 1), new Card(11, 2), new Card(11
    , 0),
216                 new Card(9, 1))));
217         String msg = "Start testing two pair vs two pair (
    first high pair card)";
218         int expected = 1;
219         int actual = hand.compareTo(other);
220 
221         Testing.assertEquals(msg, expected, actual);
222     }
223 
224     private void twoPairVsTwoPairHighCard2() {
225         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
226                 new Card(14, 1), new Card(8, 2), new Card(8,
    0),
227                 new Card(9, 1))));
228         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 1),
229                 new Card(14, 0), new Card(7, 1), new Card(7,
    2),
230                 new Card(9, 1))));
```

```
231         String msg = "Start testing two pair vs two pair (
    second high pair card)";
232         int expected = 1;
233         int actual = hand.compareTo(other);
234 
235         Testing.assertEquals(msg, expected, actual);
236     }
237 
238     private void twoPairVsTwoPairHighCard3() {
239         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 1),
240                 new Card(14, 0), new Card(8, 2), new Card(8,
    1),
241                 new Card(10, 0))));
242         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 1),
243                 new Card(14, 0), new Card(8, 1), new Card(8,
    2),
244                 new Card(9, 0))));
245         String msg = "Start testing two pair vs two pair (
    third high card)";
246         int expected = 1;
247         int actual = hand.compareTo(other);
248 
249         Testing.assertEquals(msg, expected, actual);
250     }
251 
252     private void twoPairVsTwoPairTie() {
253         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
254                 new Card(14, 1), new Card(8, 0), new Card(8,
    1),
255                 new Card(10, 0))));
256         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
257                 new Card(14, 0), new Card(8, 1), new Card(8,
    0),
258                 new Card(10, 1))));
259         String msg = "Start testing two pair vs two pair (tie
    )";
260         int expected = 0;
```

```
261         int actual = hand.compareTo(other);
262 
263         Testing.assertEquals(msg, expected, actual);
264     }
265 
266     private void pairVsHighCard() {
267         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 1),
268                 new Card(14, 0), new Card(8, 0), new Card(9,
    1),
269                 new Card(10, 0))));
270         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 1),
271                 new Card(13, 0), new Card(8, 0), new Card(9,
    1),
272                 new Card(10, 0))));
273         String msg = "Start testing pair vs high card";
274         int expected = 1;
275         int actual = hand.compareTo(other);
276 
277         Testing.assertEquals(msg, expected, actual);
278     }
279 
280     private void pairVsPairHighCard1() {
281         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
282                 new Card(14, 1), new Card(8, 0), new Card(9,
    1),
283                 new Card(11, 0))));
284         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(13, 1),
285                 new Card(13, 0), new Card(8, 1), new Card(9,
    1),
286                 new Card(10, 0))));
287         String msg = "Start testing pair vs pair (first high
    pair card)";
288         int expected = 1;
289         int actual = hand.compareTo(other);
290 
291         Testing.assertEquals(msg, expected, actual);
292     }
```

```
293
294     private void pairVsPairHighCard2() {
295         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
296                 new Card(14, 0), new Card(8, 1), new Card(9,
    1),
297                 new Card(11, 1))));
298         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
299                 new Card(14, 0), new Card(8, 1), new Card(9,
    0),
300                 new Card(10, 1))));
301         String msg = "Start testing pair vs pair (second pair
     card)";
302         int expected = 1;
303         int actual = hand.compareTo(other);
304
305         Testing.assertEquals(msg, expected, actual);
306     }
307
308     private void pairVsPairHighCard3() {
309         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
310                 new Card(14, 0), new Card(8, 1), new Card(10
    , 0),
311                 new Card(11, 1))));
312         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 1),
313                 new Card(14, 1), new Card(8, 0), new Card(9,
    0),
314                 new Card(11, 1))));
315         String msg = "Start testing pair vs pair (third pair
    card)";
316         int expected = 1;
317         int actual = hand.compareTo(other);
318
319         Testing.assertEquals(msg, expected, actual);
320     }
321
322     private void pairVsPairHighCard4() {
323         PokerHand hand = new PokerHand(new ArrayList<Card>(
```

```
323 Arrays.asList(new Card(14, 0),
324                 new Card(14, 1), new Card(9, 0), new Card(10
    , 0),
325                 new Card(11, 1))));
326         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
327                 new Card(14, 1), new Card(8, 0), new Card(10
    , 1),
328                 new Card(11, 0))));
329         String msg = "Start testing pair vs pair (fourth pair
     card)";
330         int expected = 1;
331         int actual = hand.compareTo(other);
332 
333         Testing.assertEquals(msg, expected, actual);
334     }
335 
336     private void pairVsPairTie() {
337         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
338                 new Card(14, 1), new Card(9, 1), new Card(10
    , 0),
339                 new Card(11, 0))));
340         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
341                 new Card(14, 0), new Card(9, 1), new Card(10
    , 0),
342                 new Card(11, 1))));
343         String msg = "Start testing pair vs pair (tie)";
344         int expected = 0;
345         int actual = hand.compareTo(other);
346 
347         Testing.assertEquals(msg, expected, actual);
348     }
349 
350     private void highCardVsHighCard1() {
351         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 1),
352                 new Card(13, 0), new Card(12,0), new Card(11
    , 1),
353                 new Card(10, 2))));
```

```
354         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(13, 1),
355                 new Card(12, 2), new Card(11, 1), new Card(10
    , 1),
356                 new Card(9, 2))));
357         String msg = "Start testing high card vs high card (
    first high card)";
358         int expected = 1;
359         int actual = hand.compareTo(other);
360 
361         Testing.assertEquals(msg, expected, actual);
362     }
363 
364     private void highCardVsHighCard2() {
365         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
366                 new Card(13, 1), new Card(12, 2), new Card(11
    , 0),
367                 new Card(10, 1))));
368         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
369                 new Card(12, 1), new Card(11, 2), new Card(10
    , 0),
370                 new Card(9, 1))));
371         String msg = "Start testing high card vs high card (
    second high card)";
372         int expected = 1;
373         int actual = hand.compareTo(other);
374 
375         Testing.assertEquals(msg, expected, actual);
376     }
377 
378     private void highCardVsHighCard3() {
379         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
380                 new Card(13, 1), new Card(12, 2), new Card(11
    , 0),
381                 new Card(10, 1))));
382         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
383                 new Card(13, 1), new Card(11, 2), new Card(10
```

```
383 , 0),
384                 new Card(9, 1))));
385         String msg = "Start testing high card vs high card (
    third high card)";
386         int expected = 1;
387         int actual = hand.compareTo(other);
388 
389         Testing.assertEquals(msg, expected, actual);
390     }
391 
392     private void highCardVsHighCard4() {
393         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14,0),
394                 new Card(13, 2), new Card(12, 1), new Card(11
    , 0),
395                 new Card(10, 2))));
396         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
397                 new Card(13, 2), new Card(12, 1), new Card(10
    , 0),
398                 new Card(9, 2))));
399         String msg = "Start testing high card vs high card (
    fourth high card)";
400         int expected = 1;
401         int actual = hand.compareTo(other);
402 
403         Testing.assertEquals(msg, expected, actual);
404     }
405 
406     private void highCardVsHighCard5() {
407         PokerHand hand = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
408                 new Card(13, 2), new Card(12, 1), new Card(11
    , 0),
409                 new Card(10, 2))));
410         PokerHand other = new PokerHand(new ArrayList<Card>(
    Arrays.asList(new Card(14, 0),
411                 new Card(13, 2), new Card(12, 1), new Card(11
    , 0),
412                 new Card(9, 2))));
413         String msg = "Start testing high card vs high card (
```

```
fifth high card)";
        int expected = 1;
        int actual = hand.compareTo(other);

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

    private void highCardVsHighCardTie() {
        PokerHand hand = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(14, 0),
                new Card(13, 2), new Card(12, 1), new Card(11, 0),
                new Card(10, 2))));
        PokerHand other = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(14, 0),
                new Card(13, 2), new Card(12, 1), new Card(11, 0),
                new Card(10, 2))));
        String msg = "Start testing high card vs high card (tie)";
        int expected = 0;
        int actual = hand.compareTo(other);

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

```java
/**
 * Models a collection of community cards
 */

package proj4;

import java.util.ArrayList;

public class CommunityCardSet {

    private final int MAX_COLLECTION = 5;
    private final int FIRST_CARD = 0;

    private ArrayList<Card> collection;

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

    /**
     * Adds the card to the community card set if it does not have 5 cards
     * @param card a Card object that will be added to the hand
     */
    public void addCard(Card card) {
        if (getCollection().size() < MAX_COLLECTION) {
            getCollection().add(card);
        }
    }

    /**
     * Getter for the card at the given index
     * @param index an integer greater or equal to 0
     * @return a Card object at the given index or null if index is invalid
```

```
38      */
39     public Card getIthCard(int index){
40         if (FIRST_CARD ⩽ index && index < getCollection().
   size()) {
41             return getCollection().get(index);
42         }
43 
44         else {
45             return null;
46         }
47     }
48 
49     /**
50      * Returns the readable version of the community card set
51      * @return a string for the readable version of the
   community card set
52      */
53     public String toString(){
54         String returnString = "";
55 
56         for (Card card : getCollection()) {
57             returnString += card + "\n";
58         }
59 
60         return returnString;
61     }
62 
63     /**
64      * Gets the collection of the community card set
65      * @return an array list for the collection
66      */
67     public ArrayList<Card> getCollection() {
68         return collection;
69     }
70 }
```

```java
package proj4;

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

public class StudPokerHandTester {

    private final int MAX_COLLECTION = 5;
    private final int MAX_HAND = 2;

    public static void main(String[] args) {
        StudPokerHandTester studPokerHandTester = new StudPokerHandTester();

        Testing.startTests();
        studPokerHandTester.testStudPokerHandConstructor();
        studPokerHandTester.testAddCard();
        studPokerHandTester.testGetIthCard();
        studPokerHandTester.testCompareTo();
        studPokerHandTester.testGetResult();
        Testing.finishTests();
    }

    private void testStudPokerHandConstructor() {
        Deck deck = new Deck();
        StudPokerHand hand = new StudPokerHand(new CommunityCardSet(deck.dealITimes(MAX_COLLECTION)), deck.dealITimes(MAX_HAND));
        String msg = "Start testing StudPokerHand constructor and toString method";
        String expected = "3 of Hearts\n" + "3 of Clubs\n";
        StudPokerHand actual = hand;

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

    private void testAddCard() {
        testNotFullHand();
        testFullHand();
    }
```

```
37
38      private void testNotFullHand() {
39          Deck deck = new Deck();
40          StudPokerHand hand = new StudPokerHand(new
    CommunityCardSet(deck.dealITimes(MAX_COLLECTION)), deck.
    dealITimes(MAX_HAND - 1));
41          String msg = "Start testing addCard method in the full
     hand";
42          String expected = "3 of Hearts\n" + "3 of Clubs\n";
43          hand.addCard(deck.deal());
44          StudPokerHand actual = hand;
45
46          Testing.assertEquals(msg, expected, actual.toString
    ());
47      }
48
49      private void testFullHand() {
50          Deck deck = new Deck();
51          StudPokerHand hand = new StudPokerHand(new
    CommunityCardSet(deck.dealITimes(MAX_COLLECTION)), deck.
    dealITimes(MAX_HAND - 1));
52          String msg = "Start testing addCard method in the not
    full hand";
53          String expected = "3 of Hearts\n" + "3 of Clubs\n";;
54          hand.addCard(deck.deal());
55          StudPokerHand actual = hand;
56
57          Testing.assertEquals(msg, expected, actual.toString
    ());
58      }
59
60      private void testGetIthCard() {
61          Deck deck = new Deck();
62          StudPokerHand hand = new StudPokerHand(new
    CommunityCardSet(deck.dealITimes(MAX_COLLECTION)), deck.
    dealITimes(MAX_HAND));
63          String msg = "Start testing getIthCard method";
64          String expected = "3 of Clubs";
65          Card actual = hand.getIthCard(1);
66
67          Testing.assertEquals(msg, expected, actual.toString
```

```
67 ());
68     }
69 
70     private void testCompareTo() {
71         flushVsTwoPair();
72         flushVsFlushHighCard1();
73         flushVsFlushHighCard2();
74         flushVsFlushHighCard3();
75         flushVsFlushHighCard4();
76         flushVsFlushHighCard5();
77         flushVsFlushTie();
78 
79         twoPairVsPair();
80         twoPairVsTwoPairHighCard1();
81         twoPairVsTwoPairHighCard2();
82         twoPairVsTwoPairHighCard3();
83         twoPairVsTwoPairTie();
84 
85         pairVsHighCard();
86         pairVsPairHighCard1();
87         pairVsPairHighCard2();
88         pairVsPairHighCard3();
89         pairVsPairHighCard4();
90         pairVsPairTie();
91 
92         highCardVsHighCard1();
93         highCardVsHighCard2();
94         highCardVsHighCard3();
95         highCardVsHighCard4();
96         highCardVsHighCard5();
97         highCardVsHighCardTie();
98     }
99 
100    private void flushVsTwoPair() {
101        CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(14, 1),
102                new Card(14, 1), new Card(13, 1), new Card(13
    , 1),
103                new Card(2, 0))));
104        StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(12, 1),
```

```
105                 new Card(11, 0))));
106         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(12, 0),
107                 new Card(11, 0))));
108         String msg = "Start testing flush vs two pair";
109         int expected = 1;
110         int actual = hand.compareTo(other);
111 
112         Testing.assertEquals(msg, expected, actual);
113     }
114 
115     private void flushVsFlushHighCard1() {
116         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(11, 1),
117                 new Card(10, 1), new Card(12, 1), new Card(9
    , 1),
118                 new Card(2, 0))));
119         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(14, 1),
120                 new Card(2, 0))));
121         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(13, 1),
122                 new Card(2, 0))));
123         String msg = "Start testing flush vs flush (first
    high card)";
124         int expected = 1;
125         int actual = hand.compareTo(other);
126 
127         Testing.assertEquals(msg, expected, actual);
128     }
129 
130     private void flushVsFlushHighCard2() {
131         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(14, 1),
132                 new Card(10, 1), new Card(12, 1), new Card(9
    , 1),
133                 new Card(2, 0))));
134         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(13, 1),
135                 new Card(2, 0))));
136         StudPokerHand other = new StudPokerHand(cc, new
```

```
136 ArrayList<Card>(Arrays.asList(new Card(12, 1),
137                 new Card(2, 0))));
138         String msg = "Start testing flush vs flush (second
    high card)";
139         int expected = 1;
140         int actual = hand.compareTo(other);
141 
142         Testing.assertEquals(msg, expected, actual);
143     }
144 
145     private void flushVsFlushHighCard3() {
146         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(14, 1),
147                 new Card(10, 1), new Card(12, 1), new Card(9
    , 1),
148                 new Card(2, 0))));
149         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(12, 1),
150                 new Card(2, 0))));
151         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(11, 1),
152                 new Card(2, 0))));
153         String msg = "Start testing flush vs flush (third
    high card)";
154         int expected = 1;
155         int actual = hand.compareTo(other);
156 
157         Testing.assertEquals(msg, expected, actual);
158     }
159 
160     private void flushVsFlushHighCard4() {
161         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(14, 1),
162                 new Card(13, 1), new Card(12, 1), new Card(9
    , 1),
163                 new Card(2, 0))));
164         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(12, 1),
165                 new Card(2, 0))));
166         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(11, 1),
```

```
167                 new Card(2, 0))));
168         String msg = "Start testing flush vs flush (fourth
    high card)";
169         int expected = 1;
170         int actual = hand.compareTo(other);
171 
172         Testing.assertEquals(msg, expected, actual);
173     }
174 
175     private void flushVsFlushHighCard5() {
176         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(14, 1),
177                 new Card(13, 1), new Card(12, 1), new Card(11
    , 1),
178                 new Card(2, 0))));
179         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(10, 1),
180                 new Card(2, 0))));
181         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(9, 1),
182                 new Card(2, 0))));
183         String msg = "Start testing flush vs flush (fifth
    high card)";
184         int expected = 1;
185         int actual = hand.compareTo(other);
186 
187         Testing.assertEquals(msg, expected, actual);
188     }
189 
190     private void flushVsFlushTie() {
191         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(14, 1),
192                 new Card(13, 1), new Card(12, 1), new Card(11
    , 1),
193                 new Card(2, 0))));
194         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(10, 1),
195                 new Card(2, 0))));
196         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(10, 1),
197                 new Card(2, 0))));
```

```
198         String msg = "Start testing flush vs flush (tie)";
199         int expected = 0;
200         int actual = hand.compareTo(other);
201 
202         Testing.assertEquals(msg, expected, actual);
203     }
204 
205     private void twoPairVsPair() {
206         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(14, 0),
207                 new Card(13, 1), new Card(12, 2), new Card(12
    , 1),
208                 new Card(2, 0))));
209         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(14, 1),
210                 new Card(3, 0))));
211         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(10, 1),
212                 new Card(3, 0))));
213         String msg = "Start testing two pair vs pair";
214         int expected = 1;
215         int actual = hand.compareTo(other);
216 
217         Testing.assertEquals(msg, expected, actual);
218     }
219 
220     private void twoPairVsTwoPairHighCard1() {
221         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(14, 0),
222                 new Card(13, 1), new Card(12, 2), new Card(12
    , 1),
223                 new Card(2, 0))));
224         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(14, 1),
225                 new Card(3, 0))));
226         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(13, 1),
227                 new Card(3, 0))));
228         String msg = "Start testing two pair vs two pair (
    first high card)";
229         int expected = 1;
```

```
230         int actual = hand.compareTo(other);
231 
232         Testing.assertEquals(msg, expected, actual);
233     }
234 
235     private void twoPairVsTwoPairHighCard2() {
236         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(14, 0),
237                 new Card(14, 1), new Card(13, 2), new Card(12
    , 1),
238                 new Card(2, 0))));
239         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(13, 1),
240                 new Card(3, 0))));
241         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(12, 1),
242                 new Card(3, 0))));
243         String msg = "Start testing two pair vs two pair (
    second high card)";
244         int expected = 1;
245         int actual = hand.compareTo(other);
246 
247         Testing.assertEquals(msg, expected, actual);
248     }
249 
250     private void twoPairVsTwoPairHighCard3() {
251         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(14, 0),
252                 new Card(14, 1), new Card(13, 2), new Card(3
    , 0),
253                 new Card(2, 0))));
254         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(13, 1),
255                 new Card(10, 0))));
256         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(13, 1),
257                 new Card(9, 0))));
258         String msg = "Start testing two pair vs two pair (
    third high card)";
259         int expected = 1;
260         int actual = hand.compareTo(other);
```

```
261
262         Testing.assertEquals(msg, expected, actual);
263     }
264
265     private void twoPairVsTwoPairTie() {
266         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(14, 0),
267                 new Card(14, 1), new Card(13, 2), new Card(12
    , 1),
268                 new Card(2, 0))));
269         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(13, 1),
270                 new Card(4, 0))));
271         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(13, 1),
272                 new Card(4, 0))));
273         String msg = "Start testing two pair vs two pair (tie
    )";
274         int expected = 0;
275         int actual = hand.compareTo(other);
276
277         Testing.assertEquals(msg, expected, actual);
278     }
279
280     private void pairVsHighCard() {
281         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(14, 0),
282                 new Card(13, 1), new Card(12, 2), new Card(11
    , 1),
283                 new Card(2, 0))));
284         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(14, 1),
285                 new Card(9, 0))));
286         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(10, 1),
287                 new Card(9, 0))));
288         String msg = "Start testing pair vs high card";
289         int expected = 1;
290         int actual = hand.compareTo(other);
291
292         Testing.assertEquals(msg, expected, actual);
```

```
293     }
294
295     private void pairVsPairHighCard1() {
296         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(14, 0),
297                 new Card(13, 1), new Card(12, 2), new Card(11
    , 1),
298                 new Card(2, 0))));
299         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(14, 1),
300                 new Card(9, 0))));
301         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(13, 1),
302                 new Card(9, 0))));
303         String msg = "Start testing pair vs pair (first high
    card)";
304         int expected = 1;
305         int actual = hand.compareTo(other);
306
307         Testing.assertEquals(msg, expected, actual);
308     }
309
310     private void pairVsPairHighCard2() {
311         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(10, 0),
312                 new Card(5, 1), new Card(4, 2), new Card(3, 1
    ),
313                 new Card(2, 0))));
314         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(10, 1),
315                 new Card(9, 0))));
316         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(10, 1),
317                 new Card(8, 0))));
318         String msg = "Start testing pair vs pair (second high
     card)";
319         int expected = 1;
320         int actual = hand.compareTo(other);
321
322         Testing.assertEquals(msg, expected, actual);
323     }
```

```
324
325     private void pairVsPairHighCard3() {
326         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(10, 0),
327                 new Card(9, 1), new Card(5, 2), new Card(4, 1
    ),
328                 new Card(2, 0))));
329         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(10, 1),
330                 new Card(8, 0))));
331         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(10, 1),
332                 new Card(7, 0))));
333         String msg = "Start testing pair vs pair (third high
    card)";
334         int expected = 1;
335         int actual = hand.compareTo(other);
336
337         Testing.assertEquals(msg, expected, actual);
338     }
339
340     private void pairVsPairHighCard4() {
341         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(10, 0),
342                 new Card(9, 1), new Card(7, 2), new Card(3, 1
    ),
343                 new Card(2, 0))));
344         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(10, 1),
345                 new Card(5, 0))));
346         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(10, 1),
347                 new Card(4, 0))));
348         String msg = "Start testing pair vs pair (fourth high
     card)";
349         int expected = 1;
350         int actual = hand.compareTo(other);
351
352         Testing.assertEquals(msg, expected, actual);
353     }
354
```

```
355     private void pairVsPairTie() {
356         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(10, 0),
357                 new Card(9, 1), new Card(7, 2), new Card(5, 1
    ),
358                 new Card(2, 0))));
359         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(10, 1),
360                 new Card(5, 0))));
361         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(10, 1),
362                 new Card(5, 0))));
363         String msg = "Start testing pair vs pair (tie)";
364         int expected = 0;
365         int actual = hand.compareTo(other);
366 
367         Testing.assertEquals(msg, expected, actual);
368     }
369 
370     private void highCardVsHighCard1() {
371         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(12, 0),
372                 new Card(10, 1), new Card(8, 2), new Card(6,
    1),
373                 new Card(2, 0))));
374         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(14, 1),
375                 new Card(3, 0))));
376         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(13, 1),
377                 new Card(3, 0))));
378         String msg = "Start testing high card vs high card (
    first high card)";
379         int expected = 1;
380         int actual = hand.compareTo(other);
381 
382         Testing.assertEquals(msg, expected, actual);
383     }
384 
385     private void highCardVsHighCard2() {
386         CommunityCardSet cc = new CommunityCardSet(new
```

```
386 ArrayList<Card>(Arrays.asList(new Card(12, 0),
387                 new Card(10, 1), new Card(8, 2), new Card(6,
    1),
388                 new Card(2, 0))));
389         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(11, 1),
390                 new Card(3, 0))));
391         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(9, 1),
392                 new Card(3, 0))));
393         String msg = "Start testing high card vs high card (
    second high card)";
394         int expected = 1;
395         int actual = hand.compareTo(other);
396 
397         Testing.assertEquals(msg, expected, actual);
398     }
399 
400     private void highCardVsHighCard3() {
401         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(12, 0),
402                 new Card(10, 1), new Card(8, 2), new Card(6,
    1),
403                 new Card(2, 0))));
404         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(9, 1),
405                 new Card(3, 0))));
406         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(7, 1),
407                 new Card(3, 0))));
408         String msg = "Start testing high card vs high card (
    third high card)";
409         int expected = 1;
410         int actual = hand.compareTo(other);
411 
412         Testing.assertEquals(msg, expected, actual);
413     }
414 
415     private void highCardVsHighCard4() {
416         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(12, 0),
```

```
417                 new Card(10, 1), new Card(8, 2), new Card(6,
    1),
418                 new Card(2, 0))));
419         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(7, 1),
420                 new Card(3, 0))));
421         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(5, 1),
422                 new Card(3, 0))));
423         String msg = "Start testing high card vs high card (
    fourth high card)";
424         int expected = 1;
425         int actual = hand.compareTo(other);
426 
427         Testing.assertEquals(msg, expected, actual);
428     }
429 
430     private void highCardVsHighCard5() {
431         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(12, 0),
432                 new Card(10, 1), new Card(8, 2), new Card(6,
    1),
433                 new Card(2, 0))));
434         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(5, 1),
435                 new Card(3, 0))));
436         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(4, 1),
437                 new Card(3, 0))));
438         String msg = "Start testing high card vs high card (
    fifth high card)";
439         int expected = 1;
440         int actual = hand.compareTo(other);
441 
442         Testing.assertEquals(msg, expected, actual);
443     }
444 
445     private void highCardVsHighCardTie() {
446         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(12, 0),
447                 new Card(10, 1), new Card(8, 2), new Card(6,
```

```
447 1),
448                 new Card(2, 0))));
449         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(5, 1),
450                 new Card(3, 0))));
451         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(5, 1),
452                 new Card(3, 0))));
453         String msg = "Start testing high card vs high card (
    tie)";
454         int expected = 0;
455         int actual = hand.compareTo(other);
456 
457         Testing.assertEquals(msg, expected, actual);
458     }
459 
460     private void testGetResult() {
461         testHandWinResult();
462         testOtherWinResult();
463         testTieResult();
464     }
465 
466     private void testHandWinResult() {
467         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(12, 0),
468                 new Card(10, 1), new Card(8, 2), new Card(6,
    1),
469                 new Card(2, 0))));
470         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(14, 1),
471                 new Card(3, 0))));
472         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(13, 1),
473                 new Card(3, 0))));
474         String msg = "Start testing hand winning scenario";
475         String expected = "my hand";
476         String actual = hand.getResult(other);
477 
478         Testing.assertEquals(msg, expected, actual);
479     }
480 
```

```
481     private void testOtherWinResult() {
482         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(12, 0),
483                 new Card(10, 1), new Card(8, 2), new Card(6,
    1),
484                 new Card(2, 0))));
485         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(13, 1),
486                 new Card(3, 0))));
487         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(14, 1),
488                 new Card(3, 0))));
489         String msg = "Start testing other winning scenario";
490         String expected = "other hand";
491         String actual = hand.getResult(other);
492 
493         Testing.assertEquals(msg, expected, actual);
494     }
495 
496     private void testTieResult() {
497         CommunityCardSet cc = new CommunityCardSet(new
    ArrayList<Card>(Arrays.asList(new Card(12, 0),
498                 new Card(10, 1), new Card(8, 2), new Card(6,
    1),
499                 new Card(2, 0))));
500         StudPokerHand hand = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(14, 1),
501                 new Card(3, 0))));
502         StudPokerHand other = new StudPokerHand(cc, new
    ArrayList<Card>(Arrays.asList(new Card(14, 1),
503                 new Card(3, 0))));
504         String msg = "Start testing other winning scenario";
505         String expected = "tie";
506         String actual = hand.getResult(other);
507 
508         Testing.assertEquals(msg, expected, actual);
509     }
510 }
```

```java
package proj4;

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

public class CommunityCardSetTester {

    private final int MAX_COLLECTION = 5;

    public static void main(String[] args) {
        CommunityCardSetTester communityCardSetTester = new CommunityCardSetTester();

        Testing.startTests();
        communityCardSetTester.testCommunityCardSetConstructor();
        communityCardSetTester.testAddCard();
        communityCardSetTester.testGetIthCard();
        communityCardSetTester.testGetCollection();
        Testing.finishTests();
    }

    private void testCommunityCardSetConstructor() {
        Deck deck = new Deck();
        CommunityCardSet communityCardSet = new CommunityCardSet(deck.dealITimes(MAX_COLLECTION));
        String msg = "Start testing CommunityCardSet constructor and toString method";
        String expected = "2 of Spades\n" + "2 of Hearts\n" + "2 of Clubs\n" + "2 of Diamonds\n" + "3 of Spades\n";
        CommunityCardSet actual = communityCardSet;

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

    private void testAddCard() {
        testNotFullCollection();
        testFullCollection();
    }

```

```
36      private void testNotFullCollection() {
37          Deck deck = new Deck();
38          CommunityCardSet communityCardSet = new
    CommunityCardSet(deck.dealITimes(MAX_COLLECTION - 1));
39          String msg = "Start testing addCard method in the not
    full collection";
40          String expected = "2 of Spades\n" + "2 of Hearts\n" +
    "2 of Clubs\n" + "2 of Diamonds\n" + "3 of Spades\n";
41          communityCardSet.addCard(deck.deal());
42          CommunityCardSet actual = communityCardSet;
43
44          Testing.assertEquals(msg,expected, actual.toString());
45      }
46
47      private void testFullCollection() {
48          Deck deck = new Deck();
49          CommunityCardSet communityCardSet = new
    CommunityCardSet(deck.dealITimes(MAX_COLLECTION));
50          String msg = "Start testing addCard method in the full
     collection";
51          String expected = "2 of Spades\n" + "2 of Hearts\n" +
    "2 of Clubs\n" + "2 of Diamonds\n" + "3 of Spades\n";
52          communityCardSet.addCard(deck.deal());
53          CommunityCardSet actual = communityCardSet;
54
55          Testing.assertEquals(msg,expected, actual.toString());
56      }
57
58      private void testGetIthCard() {
59          Deck deck = new Deck();
60          CommunityCardSet communityCardSet = new
    CommunityCardSet(deck.dealITimes(MAX_COLLECTION));
61          String msg = "Start testing getIthCard method";
62          String expected = "2 of Hearts";
63          Card actual = communityCardSet.getIthCard(1);
64
65          Testing.assertEquals(msg,expected, actual.toString());
66      }
67
68      private void testGetCollection() {
69          Deck deck = new Deck();
```

```
70          CommunityCardSet communityCardSet = new
    CommunityCardSet(deck.dealITimes(MAX_COLLECTION));
71          String msg = "Start testing getCollection method";
72          ArrayList<Card> expected = new ArrayList<Card>(Arrays
    .asList(new Card(2, 0),
73                  new Card(2, 1), new Card(2, 2), new Card(2, 3
    ),
74                  new Card(3, 0)));
75          ArrayList<Card> actual = communityCardSet.
    getCollection();
76
77          Testing.assertEquals(msg, expected.toString(), actual
    .toString());
78      }
79  }
```