```
1  package proj4; // do not erase. Gradescope expects this.
2  public class Card {
3      private int rank;
4      private String suit;
5
6      /**
7       * card constructor
8       * @param rank String: whole cards (2-10) can either be spelled
9       * out like "two" or numeric like "2". Case insensitive.
10      * @param suit String: "Spades", "Hearts", "Clubs", or "Diamonds"
11      */
12     public Card(String rank, String suit){
13         this.suit = suit;
14         this.rank = stringToInteger(rank);
15     }
16
17     /**
18      * card constructor
19      * @param rank integer between 2-14
20      * @param suit integer: 0=Spades, 1=Hearts, 2=Clubs, or 3=Diamonds
21      */
22     public Card(int rank, int suit){
23         this.suit = intToString(suit);
24         this.rank = rank;
25     }
26
27     /**
28      * converts all integer suit values into readable string version
29      * @param suit if suit input is an integer, fed into her
30      * @return String version of suit or null if suit input is none of possible
    integer values
31      */
32     private String intToString(int suit){
33         if (suit == 0){
34             return "Spades";
35         }
36         else if (suit == 1){
37             return "Hearts";
38         }
39         else if (suit == 2){
40             return "Clubs";
41         }
```

```
42          else if (suit == 3){
43              return "Diamonds";
44          }
45          return null;
46      }
47
48      /**
49       * converts all possible string variations of playing card rank and
    converts into corresponding integer
50       * @param rank if rank is a string, fed into here
51       * @return integer indicating rank; returns 2 as default if string does not
     fit any of possible inputs
52       */
53      private int stringToInteger(String rank) {
54          if ((rank.equalsIgnoreCase("two")) || rank.equalsIgnoreCase("2")){
55              return 2;
56          }
57          else if ((rank.equalsIgnoreCase("three")) || rank.equalsIgnoreCase("3"
    )){
58              return 3;
59          }
60          else if ((rank.equalsIgnoreCase("four")) || rank.equalsIgnoreCase("4"
    )){
61              return 4;
62          }
63          else if ((rank.equalsIgnoreCase("five")) || rank.equalsIgnoreCase("5"
    )){
64              return 5;
65          }
66          else if ((rank.equalsIgnoreCase("six")) || rank.equalsIgnoreCase("6")){
67              return 6;
68          }
69
70          else if ((rank.equalsIgnoreCase("seven")) || rank.equalsIgnoreCase("7
    ")){
71              return 7;
72          }
73          else if ((rank.equalsIgnoreCase("eight")) || rank.equalsIgnoreCase("8"
    )){
74              return 8;
75          }
76          else if ((rank.equalsIgnoreCase("nine")) || rank.equalsIgnoreCase("9"
```

```
76  )){
77              return 9;
78          }
79          else if ((rank.equalsIgnoreCase("ten")) || rank.equalsIgnoreCase("10
    "))){
80              return 10;
81          }
82          else if ((rank.equalsIgnoreCase("eleven")) || rank.equalsIgnoreCase("
    Jack") || rank.equalsIgnoreCase("11")) {
83              return 11;
84          }
85          else if ((rank.equalsIgnoreCase("twelve")) || rank.equalsIgnoreCase("
    Queen") || rank.equalsIgnoreCase("12")){
86              return 12;
87          }
88          else if ((rank.equalsIgnoreCase("thirteen")) || rank.equalsIgnoreCase
    ("King") || rank.equalsIgnoreCase("13")) {
89              return 13;
90          }
91          else if ((rank.equalsIgnoreCase("fourteen")) || rank.
    equalsIgnoreCase("Ace") || rank.equalsIgnoreCase("14")) {
92              return 14;
93          }
94          return 2;
95      }
96
97      /**
98       * returns rank value of card
99       * @return
100      */
101     public Integer getRank(){
102         return this.rank;
103     }
104
105     /**
106      * returns suit value of card
107      */
108     public String getSuit(){
109         return this.suit;
110     }
111
112     /**
```

```
113      * returns rank and suit values in readable string format
114      */
115     public String toString() {
116         String modifiedRank = Integer.toString(this.rank);
117         if (this.rank == 14){
118             modifiedRank = "Ace";
119             return modifiedRank + " of " + this.getSuit();
120         }
121         else if (this.rank == 13){
122             modifiedRank = "King";
123             return modifiedRank + " of " + this.getSuit();
124         }
125         else if (this.rank == 12){
126             modifiedRank = "Queen";
127             return modifiedRank + " of " + this.getSuit();
128         }
129         else if (this.rank == 11){
130             modifiedRank = "Jack";
131             return modifiedRank + " of " + this.getSuit();
132         }
133         return modifiedRank + " of " + this.getSuit();
134     }
135 
136 }
137 
138 
139 
```

```
1  package proj4; // do not erase. Gradescope expects this.
2  import java.util.ArrayList;
3  import java.util.concurrent.ThreadLocalRandom;
4
5  public class Deck{
6      private final String[] ranks = {"2", "3", "4", "5", "6", "7", "8", "9", "10", "
   11", "12", "13", "14"};
7      private final String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};
8      private static final int NUMBER_OF_CARDS=52;
9      private static final int NUMBER_OF_SUITS=4;
10     private static final int CARDS_IN_SUIT=13;
11     private static final int HAND_LENGTH = 5;
12     private ArrayList<Card> myDeck;
13     private int nextToDeal;
14
15     /**
16      * creates ArrayList to represent deck of cards
17      */
18     public Deck() {
19         nextToDeal = 0;
20         myDeck = new ArrayList<Card>();
21         for (String rank : ranks) {
22             for (String suit : suits) {
23                 Card card = new Card(rank, suit);
24                 myDeck.add(card);
25
26             }
27         }
28     }
29
30     /**
31      * returns shuffled deck by swapping card location with random card
   location and vice versa
32      */
33     public void shuffle() {
34         for (int oldCardIndex = nextToDeal; oldCardIndex < myDeck.size();
   oldCardIndex++) {
35             int newCardIndex = ThreadLocalRandom.current().nextInt(myDeck.
   size());
36             Card oldCard = myDeck.get(oldCardIndex);
37             Card newCard = myDeck.get(newCardIndex);
38             myDeck.set(oldCardIndex, newCard);
```

```
39              myDeck.set(newCardIndex, oldCard);
40          }
41      }
42
43      /**
44       * returns size of deck by using nextToDeal as marker of which cards
    have been dealt
45       */
46      public int size() {
47          return ((myDeck.size() - nextToDeal));
48      }
49
50      /**
51       * deals single card from created deck; progresses through deck by
    adding one to nextToDeal after every card dealt
52       * @return null if deck empty; otherwise first card in deck
53       */
54      public Card deal() {
55          if (size() > 0) {
56              Card dealtCard = myDeck.get(nextToDeal);
57              nextToDeal++;
58              return dealtCard;
59              }
60          return null;
61      }
62
63      /**
64       * gathers deck to unshuffled state by setting nextToDeal to zero again (
    moving divider between dealt and undealt
65       * cards to very front of deck
66       */
67      public void gather(){
68          nextToDeal = 0;
69      }
70
71      /**
72       * returns boolean indicating whether deck is empty or not
73       */
74      public boolean isEmpty(){
75          if (size() == 0){
76              return true;
77          }
```

```
78          return false;
79      }
80
81      /**
82       * returns readable version of deck with each card in deck made into
    string
83       */
84      public String toString(){
85          String toReturn = "";
86          for (int i = nextToDeal; i < myDeck.size(); i++){
87              Card card = myDeck.get(i);
88              toReturn = (toReturn +"|" + card.toString() + "|");
89          }
90          return toReturn;
91      }
92  }
93
```

```
1  package proj4;
2  import java.util.Scanner;
3
4  public class Main {
5      private static final int TWO_HOLE_CARD_HANDS = 4;
6      public static void main(String[] args){
7          Deck gameDeck = new Deck();
8          gameDeck.shuffle();
9          int points = 0;
10         CommunityCardSet cc = new CommunityCardSet(gameDeck);
11         while (gameDeck.size() > TWO_HOLE_CARD_HANDS){
12             StudPokerHand studHand1 = new StudPokerHand(cc, gameDeck);
13             StudPokerHand studHand2 = new StudPokerHand(cc, gameDeck);
14             System.out.println("Community Cards: " + cc);
15             System.out.println("Stud Hand 1: " + studHand1);
16             System.out.println("Stud Hand 2: " + studHand2);
17             int comparison = studHand1.compareTo(studHand2);
18             Scanner scannerObj = new Scanner(System.in);
19             System.out.println("Which hand is worth more in a game of
   poker? Type 1 for Hand1, -1 for Hand2, and 0 for a tie.");
20             String playersInput = scannerObj.nextLine();
21             if (Integer.toString(comparison).equals(playersInput)) {
22                 points++;
23                 System.out.println("Correct!");
24             }
25             if (!Integer.toString(comparison).equals(playersInput)) {
26                 System.out.println("Incorrect");
27             }
28         }
29         System.out.println("Game over! Not enough cards left in deck.
   Your score is: " + points);
30     }
31 }
32
33
```

```
1  package proj4;
2
3  /**
4   * This class contains a collection of methods that help with testing.  All
   methods
5   * here are static so there's no need to construct a Testing object.  Just call
   them
6   * with the class name like so:
7   * <p></p>
8   * <code>Testing.assertEquals("test description", expected, actual)</code>
9   *
10  * @author Kristina Striegnitz, Aaron Cass, Chris Fernandes
11  * @version 5/28/18
12  */
13 public class Testing {
14
15    private static boolean VERBOSE = false;
16    private static int numTests;
17    private static int numFails;
18
19    /**
20     * Toggles between a lot of output and little output.
21     *
22     * @param verbose
23     *           If verbose is true, then complete information is printed,
24     *           whether the tests passes or fails. If verbose is false, only
25     *           failures are printed.
26     */
27    public static void setVerbose(boolean verbose)
28    {
29       VERBOSE = verbose;
30    }
31
32    /**
33     * Each of the assertEquals methods tests whether the actual
34     * result equals the expected result. If it does, then the test
35     * passes, otherwise it fails.
36     *
37     * The only difference between these methods is the types of the
38     * parameters.
39     *
40     * All take a String message and two values of some other type to
```

```
41      * compare:
42      *
43      * @param message
44      *          a message or description of the test
45      * @param expected
46      *          the correct, or expected, value
47      * @param actual
48      *          the actual value
49      */
50     public static void assertEquals(String message, boolean expected,
51                                     boolean actual)
52     {
53         printTestCaseInfo(message, "" + expected, "" + actual);
54         if (expected == actual) {
55             pass();
56         } else {
57             fail(message);
58         }
59     }
60
61     public static void assertEquals(String message, int expected, int actual
   )
62     {
63         printTestCaseInfo(message, "" + expected, "" + actual);
64         if (expected == actual) {
65             pass();
66         } else {
67             fail(message);
68         }
69     }
70
71     public static void assertEquals(String message, Object expected,
72                                     Object actual)
73     {
74         String expectedString = "<<null>>";
75         String actualString = "<<null>>";
76         if (expected != null) {
77             expectedString = expected.toString();
78         }
79         if (actual != null) {
80             actualString = actual.toString();
81         }
```

```
82          printTestCaseInfo(message, expectedString, actualString);
83
84          if (expected == null) {
85              if (actual == null) {
86                  pass();
87              } else {
88                  fail(message);
89              }
90          } else if (expected.equals(actual)) {
91              pass();
92          } else {
93              fail(message);
94          }
95      }
96
97      /**
98       * Asserts that a given boolean must be true.  The test fails if
99       * the boolean is not true.
100      *
101      * @param message The test message
102      * @param actual The boolean value asserted to be true.
103      */
104     public static void assertTrue(String message, boolean actual)
105     {
106         assertEquals(message, true, actual);
107     }
108
109     /**
110      * Asserts that a given boolean must be false. The test fails if
111      * the boolean is not false (i.e. if it is true).
112      *
113      * @param message The test message
114      * @param actual The boolean value asserted to be false.
115      */
116     public static void assertFalse(String message, boolean actual)
117     {
118         assertEquals(message, false, actual);
119     }
120
121     private static void printTestCaseInfo(String message, String expected,
122                                           String actual)
123     {
```

```
124         if (VERBOSE) {
125             System.out.println(message + ":");
126             System.out.println("expected: " + expected);
127             System.out.println("actual:   " + actual);
128         }
129     }
130 
131     private static void pass()
132     {
133         numTests++;
134 
135         if (VERBOSE) {
136             System.out.println("--PASS--");
137             System.out.println();
138         }
139     }
140 
141     private static void fail(String description)
142     {
143         numTests++;
144         numFails++;
145 
146         if (!VERBOSE) {
147             System.out.print(description + "  ");
148         }
149         System.out.println("--FAIL--");
150         System.out.println();
151     }
152 
153     /**
154      * Prints a header for a section of tests.
155      *
156      * @param sectionTitle The header that should be printed.
157      */
158     public static void testSection(String sectionTitle)
159     {
160         if (VERBOSE) {
161             int dashCount = sectionTitle.length();
162             System.out.println(sectionTitle);
163             for (int i = 0; i < dashCount; i++) {
164                 System.out.print("-");
165             }
```

```
166             System.out.println();
167             System.out.println();
168         }
169     }
170 
171     /**
172      * Initializes the test suite. Should be called before running any
173      * tests, so that passes and fails are correctly tallied.
174      */
175     public static void startTests()
176     {
177         System.out.println("Starting Tests");
178         System.out.println();
179         numTests = 0;
180         numFails = 0;
181     }
182 
183     /**
184      * Prints out summary data at end of tests.  Should be called
185      * after all the tests have run.
186      */
187     public static void finishTests()
188     {
189         System.out.println("==============");
190         System.out.println("Tests Complete");
191         System.out.println("==============");
192         int numPasses = numTests - numFails;
193 
194         System.out.print(numPasses + "/" + numTests + " PASS ");
195         System.out.printf("(pass rate: %.1f%s)\n",
196                           100 * ((double) numPasses) / numTests,
197                           "%");
198 
199         System.out.print(numFails + "/" + numTests + " FAIL ");
200         System.out.printf("(fail rate: %.1f%s)\n",
201                           100 * ((double) numFails) / numTests,
202                           "%");
203     }
204 
205 }
206 
```

```
1  package proj4;
2
3  public class CardTests {
4      public static void main(String[] args) {
5          Testing.startTests();
6          testingIntConstructor();
7          testingIntConstructorFaceCard();
8          testingStringConstructorCase1();
9          testingStringConstructorCase2();
10         testingStringConstructorCase3();
11         testingStringConstructorFaceCard();
12         testingGetSuit();
13         testingGetRank();
14         Testing.finishTests();
15     }
16
17     public static void testingIntConstructor(){
18         Card card = new Card(4, 1);
19         String expectedCard = "4 of Hearts";
20         boolean actual = card.toString().equals(expectedCard);
21         Testing.assertEquals("Tests integer card constructor", true, actual);
22     }
23
24     public static void testingIntConstructorFaceCard(){
25         Card card = new Card(13, 3);
26         String expectedCard = "King of Diamonds";
27         boolean actual = card.toString().equals(expectedCard);
28         Testing.assertEquals("Tests integer card constructor", true, actual);
29
30     }
31
32     public static void testingStringConstructorCase1() {
33         Card card = new Card("4", "Hearts");
34         String expectedCard = "4 of Hearts";
35         boolean actual = card.toString().equals(expectedCard);
36         Testing.assertEquals("Tests string card constructor with
   numerical rank input", true, actual);
37     }
38
39     public static void testingStringConstructorCase2() {
40         Card card = new Card("four", "Hearts");
41         String expectedCard = "4 of Hearts";
```

```
42          boolean actual = card.toString().equals(expectedCard);
43          Testing.assertEquals("Tests string card constructor with rank
    spelled out input", true, actual);
44      }
45
46      public static void testingStringConstructorCase3() {
47          Card card = new Card("Four", "Hearts");
48          String expectedCard = "4 of Hearts";
49          boolean actual = card.toString().equals(expectedCard);
50          Testing.assertEquals("Tests string card constructor case
    insensitivity", true, actual);
51      }
52
53      public static void testingStringConstructorFaceCard(){
54          Card card = new Card("13", "Diamonds");
55          String expectedCard = "King of Diamonds";
56          boolean actual = card.toString().equals(expectedCard);
57          Testing.assertEquals("Tests string card constructor with face card"
    , true, actual);
58
59      }
60
61      public static void testingGetSuit(){
62          Card card = new Card("13", "Diamonds");
63          String suit = "Diamonds";
64          boolean actual = card.getSuit().equals(suit);
65          Testing.assertEquals("Tests string card constructor with face card"
    , true, actual);
66
67      }
68
69      public static void testingGetRank(){
70          Card card = new Card("Four", "Hearts");
71          int rank = 4;
72          boolean actual = card.getRank() == (rank);
73          Testing.assertEquals("Tests string card constructor with face card"
    , true, actual);
74
75      }
76
77
78
```

```
79  }
80
```

```
1  package proj4;
2
3  public class DeckTests {
4      private static final int FULL_DECK_LENGTH = 52;
5      public static void main(String[] args) {
6          Testing.startTests();
7          testingDeal();
8          testingShuffle();
9          testingSize();
10         testingGather();
11         Testing.finishTests();
12     }
13     public static void testingDeal(){
14         Deck deck = new Deck();
15         boolean expected = true;
16         String expectedCard = "2 of C";
17         Card dealtCard = deck.deal();
18         boolean actual = dealtCard.toString().equals(expectedCard);
19         Testing.assertEquals("Testing if unshuffled deck deals first card
   correctly", expected, actual);
20     }
21
22     public static void testingShuffle(){
23         Deck unshuffledDeck = new Deck();
24         Deck shuffledDeck = new Deck();
25         shuffledDeck.shuffle();
26         boolean expected = false;
27         boolean actual = unshuffledDeck.equals(shuffledDeck);
28         Testing.assertEquals("Testing if shuffled deck is different from
   unshuffled deck", expected, actual);
29     }
30
31     public static void testingSize(){
32         Deck deck = new Deck();
33         deck.deal();
34         int expected = 51;
35         int actual = deck.size();
36         Testing.assertEquals("Testing if size of deck is 51 after dealing
   one card", expected, actual);
37     }
38
39
```

```
40      public static void testingGather(){
41          Deck ungatheredDeck = new Deck();
42          Deck gatheredDeck = new Deck();
43          ungatheredDeck.deal();
44          gatheredDeck.deal();
45          gatheredDeck.gather();
46          boolean expected = false;
47          boolean actual = gatheredDeck.size() == (ungatheredDeck.size());
48          Testing.assertEquals("Tests if gathered and ungathered deck have
     different amount of cards after both dealing one card", expected,
    actual);
49      }
50
51  }
52
53
54
```

```
1  package proj4; // do not erase. Gradescope expects this.
2  import java.util.ArrayList;
3  import java.util.Collections;
4
5
6  public class PokerHand {
7      private final int HAND_LENGTH = 5;
8      private final int MAX_PAIRS = 2;
9      private ArrayList<Integer> Ranks = new ArrayList<Integer>();
10     private ArrayList<String> Suits = new ArrayList<String>();
11     private ArrayList<Integer> Pairs = new ArrayList<Integer>();
12     private ArrayList<Card> myHand = new ArrayList<Card>();
13
14     /**
15      * Creates representation of poker hand out of list of cards
16      * @param cardList: list of card objects
17      */
18     public PokerHand(ArrayList<Card> cardList) {
19         for (int i = 0; i < HAND_LENGTH; i++) {
20             Card card = cardList.get(i);
21             addCard(card);
22         }
23     }
24
25     /**
26      * Creates representation of poker hand out of deck of cards
27      * @param gameDeck: deck of card objects
28      */
29     public PokerHand(Deck gameDeck) { {
30             Card card = gameDeck.deal();
31             addCard(card);
32         }
33     }
34
35     /**
36      * checks if poker hand has standard amount of five cards, and adds
   new card only if poker hand has less than five cards
37      * @param card representation of playing card to add to hand
38      */
39     public void addCard(Card card) {
40         if (myHand.size() < HAND_LENGTH) {
41             myHand.add(card);
```

```
42          }
43      }
44
45      /**
46       * gets card from certain location of poker hand
47       * @param index indicates location of card in hand
48       * @return null if index is less than zero
49       */
50      public Card getIthCard(int index) {
51          if (index < HAND_LENGTH) {
52              return myHand.get(index);
53          }
54          return null;
55      }
56
57      /**
58       * gets ranks from all cards in poker hand and adds them to Ranks
    array list
59       * @return list of ranks
60       */
61      private ArrayList<Integer> getRanks() {
62          for (int i = 0; i < HAND_LENGTH; i++) {
63              Card card = myHand.get(i);
64              Ranks.add(card.getRank());
65          }
66          return Ranks;
67      }
68
69      /**
70       * gets suits from all cards in poker hand and adds them to Suits array
    list
71       * @return list of suits
72       */
73      private ArrayList<String> getSuits() {
74          for (int i = 0; i < HAND_LENGTH; i++) {
75              Card card = myHand.get(i);
76              Suits.add(card.getSuit());
77          }
78          return Suits;
79      }
80
81      /**
```

```
82      * determines if all cards in hand have same suit, making it a flush
83      * @return boolean value determining if hand is flush or not
84      */
85     public boolean isFlush() {
86         int flush = 1;
87         ArrayList<String> suits = getSuits();
88         for (int i = 1; i < HAND_LENGTH; i++) {
89             if (Suits.get(0).equals(Suits.get(i))) {
90                 flush++;
91                 if (flush == HAND_LENGTH) {
92                     suits.clear();
93                     return true;
94                 }
95             }
96         }
97         suits.clear();
98         return false;
99     }
100
101    /**
102     * determines if any ranks of cards in hand match, making the hand
    have a pair;
103     * if pair is found, rank of pair is found and removed twice from ranks
    list to get rid of pair;
104     * pair rank is also saved to pair list once to see what rank pair was
105     * @return modified list of ranks once pairs have been removed
106     */
107    private ArrayList<Integer> checkIfPair() {
108        for (int i = 0; i < Ranks.size(); i++) {
109            for (int j = i + 1; j < Ranks.size(); j++) {
110                if (Ranks.get(i) == Ranks.get(j)) {
111                    int pairRank = Ranks.get(i);
112                    Pairs.add(Ranks.get(i));
113                    Ranks.remove(Integer.valueOf(pairRank));
114                    Ranks.remove(Integer.valueOf(pairRank));
115                    return Ranks;
116                }
117            }
118        }
119        return Ranks;
120    }
121
```

```
122     /**
123      * getter method for pairs array list
124      * @return pairs array list
125      */
126     private ArrayList<Integer> getPairs() {
127         return Pairs;
128     }
129 
130     /**
131      * checks length of ranks from checkIfPair() to determine which type of
    pairs hand has; runs checkIfPair() again if
132      * length is 3 to check for two pairs
133      * @return string twoPair if size of ranks is 1, meaning two sets of pairs
     were removed, string onePair if size of
134      * ranks is 3, and null if size is neither 1 or 3
135      */
136     public String getPairType() {
137         getRanks();
138         checkIfPair();
139         if (Ranks.size() == 3) {
140             checkIfPair();
141             if (Ranks.size() == 1) {
142                 Ranks.clear();
143                 return "twoPair";
144             }
145             if (Ranks.size() == 3) {
146                 Ranks.clear();
147                 return "onePair";
148             }
149         }
150         return "none";
151     }
152 
153     /**
154      * sorts ranks in descending order
155      * @return first rank after sorting, meaning it returns the highest value
    rank in list
156      */
157     private int getHighestRank() {
158         Collections.sort(Ranks, Collections.reverseOrder());
159         return Ranks.get(0);
160     }
```

```
161
162     /**
163      * sorts ranks in descending order
164      * @return sorted list of ranks from highest to lowest rank
165      */
166     private ArrayList<Integer> sortRank() {
167         Collections.sort(Ranks, Collections.<Integer>reverseOrder());
168         return Ranks;
169     }
170
171     /**
172      * gets remaining ranks from list after running checkIfPair() to get highest rank value from remaining ranks;
173      * if there is more than one remaining rank, checkIfPair() is run again to check for two pairs; if remaining ranks
174      * still had more than one rank, ranks are sorted in descending order; first rank from each list of remaining ranks
175      * is compared
176      * @param other is hand to compare this hand to
177      * @return a negative number if self is worth LESS than other,
178      * zero if they are worth the SAME (a tie), and a positive number if
179      * self is worth MORE than other
180      */
181     private int getRemainingRankWinner(PokerHand other){
182         getRanks();
183         other.getRanks();
184         ArrayList<Integer> RemainingRanks1 = checkIfPair();
185         ArrayList<Integer> RemainingRanks2 = other.checkIfPair();
186         if (RemainingRanks1.size() != 1){
187             checkIfPair();
188             other.checkIfPair();
189             if (RemainingRanks1.size() != 1){
190                 Collections.sort(RemainingRanks1, Collections.<Integer>reverseOrder());
191                 Collections.sort(RemainingRanks2, Collections.<Integer>reverseOrder());
192             }
193         }
194         for (int i = 0; i < RemainingRanks1.size(); i++){
195             if (RemainingRanks1.get(i) > RemainingRanks2.get(i)){
196                 Ranks.clear();
197                 other.Ranks.clear();
```

```
198                 return 1;
199             }
200             if (RemainingRanks1.get(i) < RemainingRanks2.get(i)){
201                 Ranks.clear();
202                 other.Ranks.clear();
203                 return -1;
204             }
205         }
206         return 0;
207     }
208 
209     /**
210      * if compareTo determines both hands have same amount of pairs, gets which set of pairs has highest rank;
211      * if both sets of pairs share the same ranks, getRemainingWinner() is run to determine which of remaining non-pair
212      * ranks have highest value
213      * @param other is hand to compare this hand to
214      * @return a negative number if self is worth LESS than other,
215      * zero if they are worth the SAME (a tie), and a positive number if
216      * self is worth MORE than other
217      */
218     private int getPairsWinner(PokerHand other) {
219         ArrayList<Integer> pairs = getPairs();
220         ArrayList<Integer> otherPairs = other.getPairs();
221         ArrayList<Integer> Pairs1 = new ArrayList<Integer>();
222         ArrayList<Integer> Pairs2 = new ArrayList<Integer>();
223         Pairs1.add(pairs.get(0));
224         Pairs1.add(pairs.get(1));
225         Pairs2.add(otherPairs.get(0));
226         Pairs2.add(otherPairs.get(1));
227         Collections.sort(pairs, Collections.<Integer>reverseOrder());
228         Collections.sort(otherPairs, Collections.<Integer>reverseOrder());
229         for (int i = 0; i < MAX_PAIRS; i++) {
230             if (Pairs1.get(i) > Pairs2.get(i)) {
231                 return 1;
232             }
233             if (Pairs1.get(i) < Pairs2.get(i)) {
234                 return -1;
235             }
236         }
237         return getRemainingRankWinner(other);
```

```
238     }
239
240     /**
241      * if compareTo determines both hands are flushes, method
    determines which hand has highest rank in hand;
242      * if both hands share highest rank, next highest rank is gotten and so
    on
243      * @param other is hand to compare this hand to
244      * @return a negative number if self is worth LESS than other,
245      * zero if they are worth the SAME (a tie), and a positive number if
246      * self is worth MORE than other
247      */
248     private int getFlushesWinner(PokerHand other){
249         getRanks();
250         other.getRanks();
251         ArrayList<Integer> Ranks1 = sortRank();
252         ArrayList<Integer> Ranks2 = other.sortRank();
253         for (int i = 0; i < HAND_LENGTH; i++){
254             int HighRank1 = Ranks1.get(i);
255             int HighRank2 = Ranks2.get(i);
256             if (HighRank1 > HighRank2){
257                 Ranks1.clear();
258                 Ranks2.clear();
259                 return 1;
260             }
261             if (HighRank1 < HighRank2){
262                 Ranks1.clear();
263                 Ranks2.clear();
264                 return -1;
265             }
266         }
267         return 0;
268     }
269
270     /**
271      * if compareTo determines both hands are high card hands (neither
    are flushes or have any pairs), method gets ranks
272      * of both hands and sorts each in descending order, then
273      * gets highest rank in both hands to determine which hand is worth
    more; if both hands share highest rank value,
274      * second highest rank values are compared, and so on
275      * @param other is hand to compare this hand to
```

```
276      * @return a negative number if self is worth LESS than other,
277      *zero if they are worth the SAME (a tie), and a positive number if
278      * self is worth MORE than other
279      */
280     private int getHighCardWinner(PokerHand other){
281         Ranks.clear();
282         other.Ranks.clear();
283         getRanks();
284         sortRank();
285         other.getRanks();
286         other.sortRank();
287         for (int i = 0; i < HAND_LENGTH; i++){
288             int Rank1 = Ranks.get(i);
289             int Rank2 = other.Ranks.get(i);
290             if (Rank1 > Rank2){
291                 return 1;
292             }
293             if (Rank1 < Rank2){
294                 return -1;
295             }
296 
297         }
298         return 0;
299     }
300 
301     /**
302      * helper method for compareTo; gets results from isFlush method and
    if both hands are flushes, gets results from
303      * getFlushesWinner method
304      * @param other the hand to compare this hand to
305      * @return a negative number if this is worth LESS than other, zero
306      *if they are worth the SAME, and a positive number if this is worth
307      *MORE than other
308      */
309     private int compareToFlush(PokerHand other){
310         if (isFlush() & !other.isFlush()) {
311             return 1;
312         }
313         else if (!isFlush() & (other.isFlush())) {
314             return -1;
315         }
316         else if (isFlush() & (other.isFlush())) {
```

```
317             return getFlushesWinner(other);
318         }
319         return 0;
320     }
321 
322     /**
323      * helper method for compareTo; gets results from getPairType
    method and determines which hand has better pair type;
324      * if both have same pair type, gets results from getPairsWinner
    method
325      * @param other the hand to compare this hand to
326      * @return a negative number if this is worth LESS than other, zero
327      *if they are worth the SAME, and a positive number if this is worth
328      *MORE than other
329      */
330     private int compareToPairs(PokerHand other){
331         if ((getPairType().equals("twoPair")) & (other.getPairType().equals("
    onePair"))) {
332             return 1;
333         }
334         else if ((getPairType().equals("onePair")) & (other.getPairType().
    equals("twoPair"))) {
335             return -1;
336         }
337         else if (((getPairType().equals("onePair")) & (other.getPairType().
    equals("onePair"))) ||
338                 ((getPairType().equals("twoPair")) & (other.getPairType().equals
    ("twoPair")))) {
339             return getPairsWinner(other);
340         }
341         else if ((getPairType().equals("onePair") || (getPairType().equals("
    twoPair"))) & (other.getPairType().equals("none"))) {
342             return 1;
343         }
344         else if (getPairType().equals("none") & (other.getPairType().equals("
    onePair") || other.getPairType().equals("twoPair"))) {
345             return -1;
346         }
347         return 0;
348     }
349 
350     /**
```

```
351      * helper method for compareTo; gets results from high card winner
    method
352      * @param other the hand to compare this hand to
353      * @return a negative number if this is worth LESS than other, zero
354      *if they are worth the SAME, and a positive number if this is worth
355      *MORE than other
356      */
357     private int compareToHighCard(PokerHand other){
358         if (getHighCardWinner(other) == 1) {
359             return 1;
360         }
361         if (getHighCardWinner(other) == -1) {
362             return -1;
363         }
364         return 0;
365     }
366 
367     /**
368      * Determines how this hand compares to another hand, returns
369      * positive, negative, or zero depending on the comparison.
370      * @param other the hand to compare this hand to
371      * @return a negative number if this is worth LESS than other, zero
372      *if they are worth the SAME, and a positive number if this is worth
373      *MORE than other
374      */
375     public int compareTo(PokerHand other) {
376         if (compareToFlush(other) != 0){
377             return compareToFlush(other);
378         }
379         else if (compareToPairs(other) != 0){
380             return compareToPairs(other);
381         }
382         return compareToHighCard(other);
383     }
384 
385     /**
386      * returns readable version of hand where all cards are converted to
    strings
387      */
388     public String toString(){
389         String toReturn = "";
390         for (int i = 0; i < HAND_LENGTH; i++){
```

```
391             Card card = myHand.get(i);
392             toReturn = (toReturn + "|" + card.toString() + "|");
393         }
394         return toReturn;
395     }
396 
397 
398 
399 }
400 
```

```
1  package proj4;
2  import java.util.ArrayList;
3
4  public class StudPokerHand {
5     final int STUD_CARDS_LENGTH = 2;
6     final int HAND_LENGTH = 5;
7     final int STUD_HAND_LENGTH = 7;
8     private ArrayList<Card> communityCards = new ArrayList<Card>();
9     private ArrayList<Card> masterHand = new ArrayList<Card>();
10    private ArrayList<Card> studHand = new ArrayList<Card>();
11    private ArrayList<PokerHand> handCombos = new ArrayList<PokerHand
   >();
12       /**
13        * constructor for stud poker hand; makes three hands; a community
   card hand, a stud poker hand, and a master hand
14        * which contains both community cards and stud cards
15        * @param cc community cards
16        * @param cardList list of card objects to add to stud poker hand
17        */
18    public StudPokerHand(CommunityCardSet cc, ArrayList<Card> cardList){
19       for (int i = 0; i < HAND_LENGTH; i++){
20          communityCards.add(cc.getIthCard(i));
21          masterHand.add(cc.getIthCard(i));
22       }
23       for (int j = 0; j < STUD_CARDS_LENGTH; j++){
24          studHand.add(cardList.get(j));
25          masterHand.add(cardList.get(j));
26       }
27    }
28
29       /**
30        * constructor for stud poker hand; makes three hands; a community
   card hand, a stud poker hand, and a master hand
31        * which contains both community cards and stud cards
32        * @param cc community cards
33        * @param gameDeck deck of card objects to take two card objects to
   add to stud poker hand
34        */
35    public StudPokerHand(CommunityCardSet cc, Deck gameDeck) {
36       for (int i = 0; i < HAND_LENGTH; i++){
37          communityCards.add(cc.getIthCard(i));
38          masterHand.add(cc.getIthCard(i));
```

```
39          }
40          for (int j = 0; j < STUD_CARDS_LENGTH; j++){
41              Card cardToAdd = gameDeck.deal();
42              studHand.add(cardToAdd);
43              masterHand.add(cardToAdd);
44          }
45      }
46
47      /**
48       * adds card to stud hand if it contains less than two hole cards
49       * @param card representation of playing card to add to hand
50       */
51      public void addCard(Card card) {
52          if (studHand.size() < STUD_CARDS_LENGTH) {
53              studHand.add(card);
54          }
55      }
56
57      /**
58       * gets card from certain location of stud hand
59       * @param index indicates location of card in hand
60       * @return null if index is less than zero
61       */
62      public Card getIthCard(int index) {
63          if (index < STUD_CARDS_LENGTH) {
64
65              return studHand.get(index);
66          }
67          return null;
68      }
69
70      /**
71       * gets all possible five card poker hands that can be made from
    master hand which contains seven card hand
72       * containing both community cards and two cards in stud hand
73       * @return array list of all 21 possible poker hands made from seven
    cards
74       */
75      private ArrayList<PokerHand> getAllFiveCardHands(){
76          for (int i = 0; i < STUD_HAND_LENGTH; i++ ){
77              for (int j = i+1; j < STUD_HAND_LENGTH; j++){
78                  Card card1 = masterHand.get(i);
```

```
79                  Card card2 = masterHand.get(j);
80                  masterHand.remove(card1);
81                  masterHand.remove(card2);
82                  PokerHand hand = new PokerHand(masterHand);
83                  handCombos.add(hand);
84                  masterHand.add(i, card1);
85                  masterHand.add(j, card2);
86              }
87          }
88          return handCombos;
89      }
90
91      /**
92       * gets best possible hand out of all 21 poker hands made in
    getAllFiveCardHands method by using compareTo method
93       * in PokerHand class
94       * @return PokerHand that is the best possible poker hand out of all
    21 poker hands
95       */
96      private PokerHand getBestFiveCardHand() {
97          ArrayList<PokerHand> hands = getAllFiveCardHands();
98          PokerHand bestSoFar = hands.get(0);
99          for (int i = 1; i < hands.size(); i++) {
100             if (hands.get(i).compareTo(bestSoFar) > 0) {
101                 bestSoFar = hands.get(i);
102             }
103         }
104         return bestSoFar;
105     }
106
107     /**
108      * Determines how this hand compares to another hand, using the
109      * community card set to determine the best 5-card hand it can
110      * make. Returns positive, negative, or zero depending on the
    comparison.
111      *
112      * @param other The hand to compare this hand to
113      * @return a negative number if this is worth LESS than other, zero
114      * if they are worth the SAME, and a positive number if this is worth
115      * MORE than other
116      */
117     public int compareTo(StudPokerHand other){
```

```
118         PokerHand hand1 = getBestFiveCardHand();
119         PokerHand hand2 = other.getBestFiveCardHand();
120         return hand1.compareTo(hand2);
121     }
122 
123     /**
124      * returns readable string version of stud cards
125      */
126     public String toString(){
127         String toReturn = "";
128         for (int i = 0; i < STUD_CARDS_LENGTH; i++){
129             Card card = studHand.get(i);
130             toReturn = (toReturn + "|" + card.toString() + "|");
131         }
132         return toReturn;
133     }
134 }
135 
136 
```

```
1  /* author: Lilah Kelly
2   date: 06.01.2020
3   honor code: I affirm that I have carried out the attached academic
    endeavors with full academic honesty, in
4   accordance with the Union College Honor Code and the course syllabus.
5   */
6
7  package proj4;
8  import java.util.ArrayList;
9
10 public class PokerHandTests {
11     private static final int HAND_LENGTH = 5;
12     public static void main(String[] args) {
13         Testing.startTests();
14         testFlushHand1Wins();
15         testFlushHand2Wins();
16         testPairHand1Wins();
17         testPairHand2Wins();
18         testTwoPairHand1Wins();
19         testTwoPairHand2Wins();
20         testHighCardHand1Wins();
21         testHighCardHand2Wins();
22         testTwoFlushesHand1Wins();
23         testTwoFlushesHand2Wins();
24         testTwoTwoPairsHand1Wins();
25         testTwoTwoPairsHand2Wins();
26         testTwoPairsHand1Wins();
27         testTwoPairsHand2Wins();
28         testTwoHighCardsHand1Wins();
29         testTwoHighCardsHand2Wins();
30         testTies();
31         Testing.finishTests();
32     }
33
34     public static void testFlushHand1Wins(){
35         ArrayList<Card> cardList1 = new ArrayList<Card>();
36         ArrayList<Card> cardList2 = new ArrayList<Card>();
37         cardList1.add(new Card("4", "C"));
38         cardList1.add(new Card("5", "C"));
39         cardList1.add(new Card("6", "C"));
40         cardList1.add(new Card("7", "C"));
41         cardList1.add(new Card("8", "C"));
```

```
42          PokerHand hand1 = new PokerHand(cardList1);
43          cardList2.add(new Card("3", "C"));
44          cardList2.add(new Card("5", "H"));
45          cardList2.add(new Card("6", "C"));
46          cardList2.add(new Card("7", "C"));
47          cardList2.add(new Card("8", "C"));
48          PokerHand hand2 = new PokerHand(cardList2);
49          int actual = hand1.compareTo(hand2);
50          Testing.assertEquals("test of hand1 winning with flush", 1, actual);
51      }
52
53      public static void testFlushHand2Wins() {
54          ArrayList<Card> cardList1 = new ArrayList<Card>();
55          ArrayList<Card> cardList2 = new ArrayList<Card>();
56          cardList1.add(new Card("14", "D"));
57          cardList1.add(new Card("3", "D"));
58          cardList1.add(new Card("12", "C"));
59          cardList1.add(new Card("7", "C"));
60          cardList1.add(new Card("9", "C"));
61          PokerHand hand1 = new PokerHand(cardList1);
62          cardList2.add(new Card("8", "S"));
63          cardList2.add(new Card("9", "S"));
64          cardList2.add(new Card("10", "S"));
65          cardList2.add(new Card("11", "S"));
66          cardList2.add(new Card("12", "S"));
67          PokerHand hand2 = new PokerHand(cardList2);
68          int actual = hand1.compareTo(hand2);
69          Testing.assertEquals("test of hand2 winning with flush", -1, actual);
70      }
71
72      public static void testPairHand1Wins() {
73          ArrayList<Card> cardList1 = new ArrayList<Card>();
74          ArrayList<Card> cardList2 = new ArrayList<Card>();
75          cardList1.add(new Card("9", "H"));
76          cardList1.add(new Card("9", "C"));
77          cardList1.add(new Card("8", "D"));
78          cardList1.add(new Card("11", "C"));
79          cardList1.add(new Card("3", "S"));
80          PokerHand hand1 = new PokerHand(cardList1);
81          cardList2.add(new Card("14", "D"));
82          cardList2.add(new Card("3", "D"));
83          cardList2.add(new Card("12", "C"));
```

```
84          cardList2.add(new Card("7", "C"));
85          cardList2.add(new Card("9", "C"));
86          PokerHand hand2 = new PokerHand(cardList2);
87          int actual = hand1.compareTo(hand2);
88          Testing.assertEquals("test of hand1 winning with a pair", 1, actual
    );
89      }
90
91      public static void testPairHand2Wins() {
92          ArrayList<Card> cardList1 = new ArrayList<Card>();
93          ArrayList<Card> cardList2 = new ArrayList<Card>();
94          cardList1.add(new Card("14", "S"));
95          cardList1.add(new Card("13", "C"));
96          cardList1.add(new Card("9", "D"));
97          cardList1.add(new Card("11", "C"));
98          cardList1.add(new Card("4", "H"));
99          PokerHand hand1 = new PokerHand(cardList1);
100         cardList2.add(new Card("2", "C"));
101         cardList2.add(new Card("2", "H"));
102         cardList2.add(new Card("3", "S"));
103         cardList2.add(new Card("7", "S"));
104         cardList2.add(new Card("8", "H"));
105         PokerHand hand2 = new PokerHand(cardList2);
106         int actual = hand1.compareTo(hand2);
107         Testing.assertEquals("test of hand2 winning with a pair", -1, actual
    );
108     }
109
110     public static void testTwoPairHand1Wins() {
111         ArrayList<Card> cardList1 = new ArrayList<Card>();
112         ArrayList<Card> cardList2 = new ArrayList<Card>();
113         cardList1.add(new Card("6", "S"));
114         cardList1.add(new Card("6", "C"));
115         cardList1.add(new Card("9", "H"));
116         cardList1.add(new Card("7", "D"));
117         cardList1.add(new Card("8", "C"));
118         PokerHand hand1 = new PokerHand(cardList1);
119         cardList2.add(new Card("13", "D"));
120         cardList2.add(new Card("3", "D"));
121         cardList2.add(new Card("12", "C"));
122         cardList2.add(new Card("7", "C"));
123         cardList2.add(new Card("9", "C"));
```

```
124         PokerHand hand2 = new PokerHand(cardList2);
125         int actual = hand1.compareTo(hand2);
126         Testing.assertEquals("test of hand1 winning with two pairs", 1,
    actual);
127     }
128 
129     public static void testTwoPairHand2Wins() {
130         ArrayList<Card> cardList1 = new ArrayList<Card>();
131         ArrayList<Card> cardList2 = new ArrayList<Card>();
132         cardList1.add(new Card("2", "C"));
133         cardList1.add(new Card("4", "C"));
134         cardList1.add(new Card("7", "D"));
135         cardList1.add(new Card("13", "H"));
136         cardList1.add(new Card("8", "H"));
137         PokerHand hand1 = new PokerHand(cardList1);
138         cardList2.add(new Card("13", "C"));
139         cardList2.add(new Card("13", "D"));
140         cardList2.add(new Card("6", "H"));
141         cardList2.add(new Card("7", "S"));
142         cardList2.add(new Card("5", "D"));
143         PokerHand hand2 = new PokerHand(cardList2);
144         int actual = hand1.compareTo(hand2);
145         Testing.assertEquals("test of hand2 winning with two pairs", -1,
    actual);
146     }
147 
148     public static void testHighCardHand1Wins() {
149         ArrayList<Card> cardList1 = new ArrayList<Card>();
150         ArrayList<Card> cardList2 = new ArrayList<Card>();
151         cardList1.add(new Card("14", "D"));
152         cardList1.add(new Card("3", "D"));
153         cardList1.add(new Card("12", "C"));
154         cardList1.add(new Card("7", "C"));
155         cardList1.add(new Card("9", "C"));
156         PokerHand hand1 = new PokerHand(cardList1);
157         cardList2.add(new Card("13", "D"));
158         cardList2.add(new Card("8", "H"));
159         cardList2.add(new Card("11", "C"));
160         cardList2.add(new Card("6", "C"));
161         cardList2.add(new Card("4", "C"));
162         PokerHand hand2 = new PokerHand(cardList2);
163         int actual = hand1.compareTo(hand2);
```

```
164         Testing.assertEquals("test of hand1 winning with high card", 1,
    actual);
165     }
166
167     public static void testHighCardHand2Wins() {
168         ArrayList<Card> cardList1 = new ArrayList<Card>();
169         ArrayList<Card> cardList2 = new ArrayList<Card>();
170         cardList1.add(new Card("13", "C"));
171         cardList1.add(new Card("7", "H"));
172         cardList1.add(new Card("6", "C"));
173         cardList1.add(new Card("3", "S"));
174         cardList1.add(new Card("8", "C"));
175         PokerHand hand1 = new PokerHand(cardList1);
176         cardList2.add(new Card("4", "C"));
177         cardList2.add(new Card("9", "H"));
178         cardList2.add(new Card("3", "D"));
179         cardList2.add(new Card("7", "C"));
180         cardList2.add(new Card("9", "C"));
181         PokerHand hand2 = new PokerHand(cardList2);
182         int actual = hand1.compareTo(hand2);
183         Testing.assertEquals("test of hand2 winning with high card", -1,
    actual);
184     }
185
186     public static void testTwoFlushesHand1Wins() {
187         ArrayList<Card> cardList1 = new ArrayList<Card>();
188         ArrayList<Card> cardList2 = new ArrayList<Card>();
189         cardList1.add(new Card("4", "C"));
190         cardList1.add(new Card("5", "C"));
191         cardList1.add(new Card("6", "C"));
192         cardList1.add(new Card("7", "C"));
193         cardList1.add(new Card("8", "C"));
194         PokerHand hand1 = new PokerHand(cardList1);
195         cardList2.add(new Card("2", "H"));
196         cardList2.add(new Card("4", "H"));
197         cardList2.add(new Card("4", "H"));
198         cardList2.add(new Card("5", "H"));
199         cardList2.add(new Card("6", "H"));
200         PokerHand hand2 = new PokerHand(cardList2);
201         int actual = hand1.compareTo(hand2);
202         Testing.assertEquals("test of hand1 winning if both hands were
    flushes", 1, actual);
```

```
203     }
204
205     public static void testTwoFlushesHand2Wins() {
206         ArrayList<Card> cardList1 = new ArrayList<Card>();
207         ArrayList<Card> cardList2 = new ArrayList<Card>();
208         cardList1.add(new Card("4", "C"));
209         cardList1.add(new Card("5", "C"));
210         cardList1.add(new Card("6", "C"));
211         cardList1.add(new Card("7", "C"));
212         cardList1.add(new Card("8", "C"));
213         PokerHand hand1 = new PokerHand(cardList1);
214         cardList2.add(new Card("8", "H"));
215         cardList2.add(new Card("9", "H"));
216         cardList2.add(new Card("10", "H"));
217         cardList2.add(new Card("11", "H"));
218         cardList2.add(new Card("12", "H"));
219         PokerHand hand2 = new PokerHand(cardList2);
220         int actual = hand1.compareTo(hand2);
221         Testing.assertEquals("test of hand2 winning if both hands were
    flushes", -1, actual);
222     }
223
224     public static void testTwoTwoPairsHand1Wins() {
225         ArrayList<Card> cardList1 = new ArrayList<Card>();
226         ArrayList<Card> cardList2 = new ArrayList<Card>();
227         cardList1.add(new Card("6", "C"));
228         cardList1.add(new Card("6", "D"));
229         cardList1.add(new Card("13", "S"));
230         cardList1.add(new Card("13", "H"));
231         cardList1.add(new Card("2", "D"));
232         PokerHand hand1 = new PokerHand(cardList1);
233         cardList2.add(new Card("4", "C"));
234         cardList2.add(new Card("4", "H"));
235         cardList2.add(new Card("10", "D"));
236         cardList2.add(new Card("10", "C"));
237         cardList2.add(new Card("8", "C"));
238         PokerHand hand2 = new PokerHand(cardList2);
239         int actual = hand1.compareTo(hand2);
240         Testing.assertEquals("test of hand1 winning if both hands had
    two pairs", 1, actual);
241     }
242
```

```
243     public static void testTwoTwoPairsHand2Wins() {
244         ArrayList<Card> cardList1 = new ArrayList<Card>();
245         ArrayList<Card> cardList2 = new ArrayList<Card>();
246         cardList1.add(new Card("2", "C"));
247         cardList1.add(new Card("2", "H"));
248         cardList1.add(new Card("7", "C"));
249         cardList1.add(new Card("7", "S"));
250         cardList1.add(new Card("8", "C"));
251         PokerHand hand1 = new PokerHand(cardList1);
252         cardList2.add(new Card("5", "C"));
253         cardList2.add(new Card("5", "H"));
254         cardList2.add(new Card("7", "C"));
255         cardList2.add(new Card("7", "S"));
256         cardList2.add(new Card("9", "H"));
257         PokerHand hand2 = new PokerHand(cardList2);
258         int actual = hand1.compareTo(hand2);
259         Testing.assertEquals("test of hand2 winning if both hands had
    two pairs", -1, actual);
260     }
261 
262     public static void testTwoPairsHand1Wins() {
263         ArrayList<Card> cardList1 = new ArrayList<Card>();
264         ArrayList<Card> cardList2 = new ArrayList<Card>();
265         cardList1.add(new Card("9", "H"));
266         cardList1.add(new Card("9", "C"));
267         cardList1.add(new Card("8", "D"));
268         cardList1.add(new Card("11", "C"));
269         cardList1.add(new Card("3", "S"));
270         PokerHand hand1 = new PokerHand(cardList1);
271         cardList2.add(new Card("7", "C"));
272         cardList2.add(new Card("7", "S"));
273         cardList2.add(new Card("6", "D"));
274         cardList2.add(new Card("7", "H"));
275         cardList2.add(new Card("9", "C"));
276         PokerHand hand2 = new PokerHand(cardList2);
277         int actual = hand1.compareTo(hand2);
278         Testing.assertEquals("test of hand1 winning if both hands had a
    pair", 1, actual);
279     }
280 
281     public static void testTwoPairsHand2Wins() {
282         ArrayList<Card> cardList1 = new ArrayList<Card>();
```

```
283         ArrayList<Card> cardList2 = new ArrayList<Card>();
284         cardList1.add(new Card("4", "C"));
285         cardList1.add(new Card("4", "S"));
286         cardList1.add(new Card("8", "D"));
287         cardList1.add(new Card("10", "C"));
288         cardList1.add(new Card("11", "H"));
289         PokerHand hand1 = new PokerHand(cardList1);
290         cardList2.add(new Card("2", "C"));
291         cardList2.add(new Card("12", "H"));
292         cardList2.add(new Card("12", "C"));
293         cardList2.add(new Card("13", "C"));
294         cardList2.add(new Card("14", "C"));
295         PokerHand hand2 = new PokerHand(cardList2);
296         int actual = hand1.compareTo(hand2);
297         Testing.assertEquals("test of hand2 winning if both hands had a
    pair", -1, actual);
298     }
299 
300     public static void testTwoHighCardsHand1Wins() {
301         ArrayList<Card> cardList1 = new ArrayList<Card>();
302         ArrayList<Card> cardList2 = new ArrayList<Card>();
303         cardList1.add(new Card("14", "D"));
304         cardList1.add(new Card("3", "D"));
305         cardList1.add(new Card("12", "C"));
306         cardList1.add(new Card("7", "C"));
307         cardList1.add(new Card("9", "C"));
308         PokerHand hand1 = new PokerHand(cardList1);
309         cardList2.add(new Card("12", "D"));
310         cardList2.add(new Card("4", "H"));
311         cardList2.add(new Card("6", "S"));
312         cardList2.add(new Card("7", "H"));
313         cardList2.add(new Card("10", "D"));
314         PokerHand hand2 = new PokerHand(cardList2);
315         int actual = hand1.compareTo(hand2);
316         Testing.assertEquals("test of hand1 winning if both were high
    card hands", 1, actual);
317     }
318     public static void testTwoHighCardsHand2Wins() {
319         ArrayList<Card> cardList1 = new ArrayList<Card>();
320         ArrayList<Card> cardList2 = new ArrayList<Card>();
321         cardList1.add(new Card("12", "D"));
322         cardList1.add(new Card("3", "H"));
```

```
323         cardList1.add(new Card("10", "C"));
324         cardList1.add(new Card("7", "C"));
325         cardList1.add(new Card("9", "C"));
326         PokerHand hand1 = new PokerHand(cardList1);
327         cardList2.add(new Card("14", "D"));
328         cardList2.add(new Card("3", "D"));
329         cardList2.add(new Card("6", "S"));
330         cardList2.add(new Card("7", "H"));
331         cardList2.add(new Card("10", "D"));
332         PokerHand hand2 = new PokerHand(cardList2);
333         int actual = hand1.compareTo(hand2);
334         Testing.assertEquals("test of hand2 winning if both were high
    card hands", -1, actual);
335     }
336     public static void testTies() {
337         ArrayList<Card> cardList1 = new ArrayList<Card>();
338         ArrayList<Card> cardList2 = new ArrayList<Card>();
339         cardList1.add(new Card( "2", "H"));
340         cardList1.add(new Card("3", "C"));
341         cardList1.add(new Card("4", "S"));
342         cardList1.add(new Card("5", "C"));
343         cardList1.add(new Card("6", "C"));
344         PokerHand hand1 = new PokerHand(cardList1);
345         cardList2.add(new Card("2", "C"));
346         cardList2.add(new Card("3", "H"));
347         cardList2.add(new Card("4", "D"));
348         cardList2.add(new Card("5", "H"));
349         cardList2.add(new Card("6", "S"));
350         PokerHand hand2 = new PokerHand(cardList2);
351         int actual = hand1.compareTo(hand2);
352         Testing.assertEquals("test of tie between two hands", 0, actual);
353     }
354 }
355
356
```

```java
package proj4;
import java.util.ArrayList;

public class CommunityCardSet {
    final int HAND_LENGTH = 5;
    private ArrayList<Card> communityHand = new ArrayList<Card>();

    /**
     * constructor for Community Card Set
     * @param cardList takes list of Cards to add to Community Card Set
     */
    public CommunityCardSet(ArrayList<Card> cardList){
        for (int i = 0; i < cardList.size(); i++) {
            Card card = cardList.get(i);
            communityHand.add(card);
        }
    }

    /**
     * constructor for Community Card Set
     * @param gameDeck takes deck of cards and adds first five to Community Card Set
     */
    public CommunityCardSet(Deck gameDeck) {
        for (int i = 0; i < HAND_LENGTH; i++){
            Card card = gameDeck.deal();
            communityHand.add(card);
        }
    }

    /**
     * checks if poker hand has standard amount of five cards, and adds new card only if poker hand has less than five cards
     * @param card representation of playing card to add to hand
     */
    public void addCard(Card card) {
        if (communityHand.size() < HAND_LENGTH) {
            communityHand.add(card);
        }
    }

    /**
```

```
41      * gets card from certain location of poker hand
42      * @param index indicates location of card in hand
43      * @return null if index is less than zero
44      */
45     public Card getIthCard(int index) {
46         if (index < HAND_LENGTH) {
47             return communityHand.get(index);
48         }
49         return null;
50     }
51 
52     /**
53      * returns readable string version of community hand
54      */
55     public String toString(){
56         String toReturn = "";
57         for (int i = 0; i < HAND_LENGTH; i++){
58             Card card = communityHand.get(i);
59             toReturn = (toReturn + "|" + card.toString() + "|");
60         }
61         return toReturn;
62     }
63 
64 
65 }
66 
```

```
1  package proj4;
2  import java.util.ArrayList;
3
4  public class CommunityCardTests {
5      private static final int COMMMUNITY_HAND_LENGTH = 5;
6      public static void main(String[] args) {
7          Testing.startTests();
8          testingAddCard();
9          testingNoAddCard();
10         Testing.finishTests();
11     }
12
13     public static void testingAddCard(){
14         ArrayList<Card> cardList = new ArrayList<Card>();
15         cardList.add(new Card("4", "C"));
16         cardList.add(new Card("5", "C"));
17         cardList.add(new Card("6", "H"));
18         cardList.add(new Card("7", "S"));
19         CommunityCardSet cc = new CommunityCardSet(cardList);
20         Card toAdd = new Card("8", "D");
21         cc.addCard(toAdd);
22         Card lastCardInCC= cc.getIthCard(COMMMUNITY_HAND_LENGTH-1);
23         boolean actual = lastCardInCC.equals(toAdd);
24         Testing.assertEquals("Testing if add card function adds card to
   end of community card hand when only starting with four cards",
   true, actual);
25     }
26
27     public static void testingNoAddCard(){
28         ArrayList<Card> cardList = new ArrayList<Card>();
29         cardList.add(new Card("4", "C"));
30         cardList.add(new Card("5", "C"));
31         cardList.add(new Card("6", "H"));
32         cardList.add(new Card("7", "S"));
33         cardList.add(new Card("9", "C"));
34         CommunityCardSet cc = new CommunityCardSet(cardList);
35         Card toAdd = new Card("8", "D");
36         cc.addCard(toAdd);
37         Card lastCardInCC= cc.getIthCard(COMMMUNITY_HAND_LENGTH-1);
38         boolean actual = lastCardInCC.equals(toAdd);
39         Testing.assertEquals("Testing if add card function stops adding
   new cards if Community Card Set already has five cards", false, actual
```

```
39  );
40      }
41
42  }
43
```

```
1  package proj4;
2
3  import java.util.ArrayList;
4
5  public class StudPokerHandTests {
6     private static final int FULL_DECK_LENGTH = 52;
7     private static final int STUD_HAND_LENGTH = 2;
8
9     public static void main(String[] args) {
10       Testing.startTests();
11       testFlushHand1Wins();
12       testFlushHand2Wins();
13       testPairHand1Wins();
14       testPairHand2Wins();
15       testTwoPairHand1Wins();
16       testTwoPairHand2Wins();
17       testHighCardHand1Wins();
18       testHighCardHand2Wins();
19       testTwoFlushesHand1Wins();
20       testTwoFlushesHand2Wins();
21       testTwoPairsHand1Wins();
22       testTwoPairsHand2Wins();
23       testTwoTwoPairsHand1Wins();
24       testTwoTwoPairsHand2Wins();
25       testTwoHighCardsHand1Wins();
26       testTwoHighCardsHand2Wins();
27       testTies();
28       testTies2();
29       testPairHand2Wins();
30       testHighCardHand1Wins();
31       testHighCardHand2Wins();
32       Testing.finishTests();
33    }
34
35    public static void testFlushHand1Wins() {
36       ArrayList<Card> cardList = new ArrayList<Card>();
37       ArrayList<Card> hand1 = new ArrayList<Card>();
38       ArrayList<Card> hand2 = new ArrayList<Card>();
39       cardList.add(new Card("4", "C"));
40       cardList.add(new Card("5", "C"));
41       cardList.add(new Card("6", "C"));
42       cardList.add(new Card("7", "C"));
```

```
43          cardList.add(new Card("8", "C"));
44          CommunityCardSet cc = new CommunityCardSet(cardList);
45          hand1.add(new Card("3", "H"));
46          hand1.add(new Card("King", "C"));
47          hand2.add(new Card(4, 0));
48          hand2.add(new Card(5, 1));
49          StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
50          StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
51          int actual = studHand1.compareTo(studHand2);
52          Testing.assertEquals("test of hand1 winning with flush", 1, actual);
53      }
54
55      public static void testFlushHand2Wins() {
56          ArrayList<Card> cardList = new ArrayList<Card>();
57          ArrayList<Card> hand1 = new ArrayList<Card>();
58          ArrayList<Card> hand2 = new ArrayList<Card>();
59          cardList.add(new Card("4", "C"));
60          cardList.add(new Card("5", "C"));
61          cardList.add(new Card("6", "C"));
62          cardList.add(new Card("7", "C"));
63          cardList.add(new Card("8", "C"));
64          CommunityCardSet cc = new CommunityCardSet(cardList);
65          hand1.add(new Card(2, 1));
66          hand1.add(new Card(3, 0));
67          hand2.add(new Card("King", "H"));
68          hand2.add(new Card("Queen", "C"));
69          StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
70          StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
71          int actual = studHand1.compareTo(studHand2);
72          Testing.assertEquals("test of hand2 winning with flush", -1, actual);
73      }
74
75      public static void testPairHand1Wins() {
76          ArrayList<Card> cardList = new ArrayList<Card>();
77          ArrayList<Card> hand1 = new ArrayList<Card>();
78          ArrayList<Card> hand2 = new ArrayList<Card>();
79          cardList.add(new Card("9", "H"));
80          cardList.add(new Card("7", "D"));
81          cardList.add(new Card("8", "C"));
82          cardList.add(new Card("11", "C"));
83          cardList.add(new Card("3", "S"));
84          CommunityCardSet cc = new CommunityCardSet(cardList);
```

```
85          hand1.add(new Card("14", "H"));
86          hand1.add(new Card("7", "D"));
87          hand2.add(new Card("12", "C"));
88          hand2.add(new Card("6", "S"));
89          StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
90          StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
91          int actual = studHand1.compareTo(studHand2);
92          Testing.assertEquals("test of hand1 winning with a pair", 1, actual
    );
93      }
94      public static void testPairHand2Wins() {
95          ArrayList<Card> cardList = new ArrayList<Card>();
96          ArrayList<Card> hand1 = new ArrayList<Card>();
97          ArrayList<Card> hand2 = new ArrayList<Card>();
98          cardList.add(new Card("6", "H"));
99          cardList.add(new Card("9", "D"));
100         cardList.add(new Card("8", "C"));
101         cardList.add(new Card("3", "C"));
102         cardList.add(new Card("10", "S"));
103         CommunityCardSet cc = new CommunityCardSet(cardList);
104         hand1.add(new Card("14", "H"));
105         hand1.add(new Card("7", "D"));
106         hand2.add(new Card("12", "C"));
107         hand2.add(new Card("6", "S"));
108         StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
109         StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
110         int actual = studHand1.compareTo(studHand2);
111         Testing.assertEquals("test of hand2 winning with a pair", -1, actual
    );
112     }
113     public static void testTwoPairHand1Wins() {
114         ArrayList<Card> cardList = new ArrayList<Card>();
115         ArrayList<Card> hand1 = new ArrayList<Card>();
116         ArrayList<Card> hand2 = new ArrayList<Card>();
117         cardList.add(new Card("9", "H"));
118         cardList.add(new Card("7", "D"));
119         cardList.add(new Card("8", "C"));
120         cardList.add(new Card("11", "C"));
121         cardList.add(new Card("3", "S"));
122         CommunityCardSet cc = new CommunityCardSet(cardList);
123         hand1.add(new Card("11", "H"));
124         hand1.add(new Card("7", "D"));
```

```
125         hand2.add(new Card("12", "C"));
126         hand2.add(new Card("6", "S"));
127         StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
128         StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
129         int actual = studHand1.compareTo(studHand2);
130         Testing.assertEquals("test of hand1 winning with two pairs", 1,
    actual);
131     }
132     public static void testTwoPairHand2Wins() {
133         ArrayList<Card> cardList = new ArrayList<Card>();
134         ArrayList<Card> hand1 = new ArrayList<Card>();
135         ArrayList<Card> hand2 = new ArrayList<Card>();
136         cardList.add(new Card("3", "H"));
137         cardList.add(new Card("8", "D"));
138         cardList.add(new Card("13", "C"));
139         cardList.add(new Card("11", "C"));
140         cardList.add(new Card("3", "S"));
141         CommunityCardSet cc = new CommunityCardSet(cardList);
142         hand1.add(new Card("14", "H"));
143         hand1.add(new Card("7", "D"));
144         hand2.add(new Card("11", "H"));
145         hand2.add(new Card("6", "S"));
146         StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
147         StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
148         int actual = studHand1.compareTo(studHand2);
149         Testing.assertEquals("test of hand2 winning with two pairs", -1,
    actual);
150     }
151     public static void testHighCardHand1Wins() {
152         ArrayList<Card> cardList = new ArrayList<Card>();
153         ArrayList<Card> hand1 = new ArrayList<Card>();
154         ArrayList<Card> hand2 = new ArrayList<Card>();
155         cardList.add(new Card("9", "H"));
156         cardList.add(new Card("7", "D"));
157         cardList.add(new Card("8", "C"));
158         cardList.add(new Card("11", "C"));
159         cardList.add(new Card("3", "S"));
160         CommunityCardSet cc = new CommunityCardSet(cardList);
161         hand1.add(new Card("14", "H"));
162         hand1.add(new Card("2", "D"));
163         hand2.add(new Card("12", "C"));
164         hand2.add(new Card("6", "S"));
```

```
165         StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
166         StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
167         int actual = studHand1.compareTo(studHand2);
168         Testing.assertEquals("test of hand1 winning with high card", 1,
    actual);
169     }
170     public static void testHighCardHand2Wins() {
171         ArrayList<Card> cardList = new ArrayList<Card>();
172         ArrayList<Card> hand1 = new ArrayList<Card>();
173         ArrayList<Card> hand2 = new ArrayList<Card>();
174         cardList.add(new Card("10", "H"));
175         cardList.add(new Card("9", "D"));
176         cardList.add(new Card("8", "C"));
177         cardList.add(new Card("11", "C"));
178         cardList.add(new Card("6", "S"));
179         CommunityCardSet cc = new CommunityCardSet(cardList);
180         hand1.add(new Card("4", "H"));
181         hand1.add(new Card("7", "D"));
182         hand2.add(new Card("12", "C"));
183         hand2.add(new Card("5", "S"));
184         StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
185         StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
186         int actual = studHand1.compareTo(studHand2);
187         Testing.assertEquals("test of hand2 winning with high card", -1,
    actual);
188 
189     }
190     public static void testTwoFlushesHand1Wins() {
191         ArrayList<Card> cardList = new ArrayList<Card>();
192         ArrayList<Card> hand1 = new ArrayList<Card>();
193         ArrayList<Card> hand2 = new ArrayList<Card>();
194         cardList.add(new Card("9", "H"));
195         cardList.add(new Card("7", "C"));
196         cardList.add(new Card("8", "C"));
197         cardList.add(new Card("11", "C"));
198         cardList.add(new Card("3", "S"));
199         CommunityCardSet cc = new CommunityCardSet(cardList);
200         hand1.add(new Card("14", "C"));
201         hand1.add(new Card("8", "C"));
202         hand2.add(new Card("12", "C"));
203         hand2.add(new Card("6", "C"));
204         StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
```

```
205         StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
206         int actual = studHand1.compareTo(studHand2);
207         Testing.assertEquals("test of hand1 winning if both hands were
    flushes", 1, actual);
208     }
209     public static void testTwoFlushesHand2Wins() {
210         ArrayList<Card> cardList = new ArrayList<Card>();
211         ArrayList<Card> hand1 = new ArrayList<Card>();
212         ArrayList<Card> hand2 = new ArrayList<Card>();
213         cardList.add(new Card("9", "H"));
214         cardList.add(new Card("7", "H"));
215         cardList.add(new Card("8", "H"));
216         cardList.add(new Card("11", "C"));
217         cardList.add(new Card("3", "S"));
218         CommunityCardSet cc = new CommunityCardSet(cardList);
219         hand1.add(new Card("4", "H"));
220         hand1.add(new Card("7", "H"));
221         hand2.add(new Card("12", "H"));
222         hand2.add(new Card("6", "H"));
223         StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
224         StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
225         int actual = studHand1.compareTo(studHand2);
226         Testing.assertEquals("test of hand2 winning if both hands were
    flushes", -1, actual);
227     }
228     public static void testTwoTwoPairsHand1Wins() {
229         ArrayList<Card> cardList = new ArrayList<Card>();
230         ArrayList<Card> hand1 = new ArrayList<Card>();
231         ArrayList<Card> hand2 = new ArrayList<Card>();
232         cardList.add(new Card("9", "H"));
233         cardList.add(new Card("9", "D"));
234         cardList.add(new Card("8", "C"));
235         cardList.add(new Card("11", "C"));
236         cardList.add(new Card("3", "S"));
237         CommunityCardSet cc = new CommunityCardSet(cardList);
238         hand1.add(new Card("14", "H"));
239         hand1.add(new Card("14", "D"));
240         hand2.add(new Card("12", "C"));
241         hand2.add(new Card("12", "S"));
242         StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
243         StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
244         int actual = studHand1.compareTo(studHand2);
```

```
245         Testing.assertEquals("test of hand1 winning if both hands had
    two pairs", 1, actual);
246     }
247     public static void testTwoTwoPairsHand2Wins() {
248         ArrayList<Card> cardList = new ArrayList<Card>();
249         ArrayList<Card> hand1 = new ArrayList<Card>();
250         ArrayList<Card> hand2 = new ArrayList<Card>();
251         cardList.add(new Card("9", "H"));
252         cardList.add(new Card("7", "D"));
253         cardList.add(new Card("8", "C"));
254         cardList.add(new Card("11", "C"));
255         cardList.add(new Card("3", "S"));
256         CommunityCardSet cc = new CommunityCardSet(cardList);
257         hand1.add(new Card("7", "H"));
258         hand1.add(new Card("9", "D"));
259         hand2.add(new Card("11", "S"));
260         hand2.add(new Card("9", "S"));
261         StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
262         StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
263         int actual = studHand1.compareTo(studHand2);
264         Testing.assertEquals("test of hand2 winning if both hands had
    two pairs", -1, actual);
265     }
266     public static void testTwoPairsHand1Wins() {
267         ArrayList<Card> cardList = new ArrayList<Card>();
268         ArrayList<Card> hand1 = new ArrayList<Card>();
269         ArrayList<Card> hand2 = new ArrayList<Card>();
270         cardList.add(new Card("9", "H"));
271         cardList.add(new Card("7", "D"));
272         cardList.add(new Card("8", "C"));
273         cardList.add(new Card("11", "C"));
274         cardList.add(new Card("3", "S"));
275         CommunityCardSet cc = new CommunityCardSet(cardList);
276         hand1.add(new Card("14", "H"));
277         hand1.add(new Card("7", "S"));
278         hand2.add(new Card("12", "C"));
279         hand2.add(new Card("7", "C"));
280         StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
281         StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
282         int actual = studHand1.compareTo(studHand2);
283         Testing.assertEquals("test of hand1 winning if both hands had a
    pair", 1, actual);
```

```
284     }
285     public static void testTwoPairsHand2Wins() {
286         ArrayList<Card> cardList = new ArrayList<Card>();
287         ArrayList<Card> hand1 = new ArrayList<Card>();
288         ArrayList<Card> hand2 = new ArrayList<Card>();
289         cardList.add(new Card("9", "H"));
290         cardList.add(new Card("7", "D"));
291         cardList.add(new Card("10", "C"));
292         cardList.add(new Card("11", "C"));
293         cardList.add(new Card("13", "S"));
294         CommunityCardSet cc = new CommunityCardSet(cardList);
295         hand1.add(new Card("14", "H"));
296         hand1.add(new Card("7", "D"));
297         hand2.add(new Card("12", "C"));
298         hand2.add(new Card("9", "S"));
299         StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
300         StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
301         int actual = studHand1.compareTo(studHand2);
302         Testing.assertEquals("test of hand2 winning if both hands had a
    pair", -1, actual);
303     }
304     public static void testTwoHighCardsHand1Wins() {
305         ArrayList<Card> cardList = new ArrayList<Card>();
306         ArrayList<Card> hand1 = new ArrayList<Card>();
307         ArrayList<Card> hand2 = new ArrayList<Card>();
308         cardList.add(new Card("9", "H"));
309         cardList.add(new Card("7", "D"));
310         cardList.add(new Card("8", "C"));
311         cardList.add(new Card("11", "C"));
312         cardList.add(new Card("3", "S"));
313         CommunityCardSet cc = new CommunityCardSet(cardList);
314         hand1.add(new Card("14", "H"));
315         hand1.add(new Card("5", "D"));
316         hand2.add(new Card("12", "C"));
317         hand2.add(new Card("6", "S"));
318         StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
319         StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
320         int actual = studHand1.compareTo(studHand2);
321         Testing.assertEquals("test of hand1 winning if both were high
    card hands", 1, actual);
322     }
323     public static void testTwoHighCardsHand2Wins() {
```

```
324         ArrayList<Card> cardList = new ArrayList<Card>();
325         ArrayList<Card> hand1 = new ArrayList<Card>();
326         ArrayList<Card> hand2 = new ArrayList<Card>();
327         cardList.add(new Card("1", "H"));
328         cardList.add(new Card("2", "D"));
329         cardList.add(new Card("3", "C"));
330         cardList.add(new Card("4", "C"));
331         cardList.add(new Card("5", "S"));
332         CommunityCardSet cc = new CommunityCardSet(cardList);
333         hand1.add(new Card("13", "H"));
334         hand1.add(new Card("7", "D"));
335         hand2.add(new Card("14", "C"));
336         hand2.add(new Card("6", "S"));
337         StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
338         StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
339         int actual = studHand1.compareTo(studHand2);
340         Testing.assertEquals("test of hand2 winning if both were high
    card hands", -1, actual);
341     }
342     public static void testTies() {
343         ArrayList<Card> cardList = new ArrayList<Card>();
344         ArrayList<Card> hand1 = new ArrayList<Card>();
345         ArrayList<Card> hand2 = new ArrayList<Card>();
346         cardList.add(new Card("9", "H"));
347         cardList.add(new Card("7", "D"));
348         cardList.add(new Card("8", "C"));
349         cardList.add(new Card("11", "C"));
350         cardList.add(new Card("10", "S"));
351         CommunityCardSet cc = new CommunityCardSet(cardList);
352         hand1.add(new Card("4", "H"));
353         hand1.add(new Card("5", "D"));
354         hand2.add(new Card("2", "C"));
355         hand2.add(new Card("6", "S"));
356         StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
357         StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
358         int actual = studHand1.compareTo(studHand2);
359         Testing.assertEquals("test of tie using all community cards
    between two hands", 0, actual);
360     }
361     public static void testTies2() {
362         ArrayList<Card> cardList = new ArrayList<Card>();
363         ArrayList<Card> hand1 = new ArrayList<Card>();
```

```
364         ArrayList<Card> hand2 = new ArrayList<Card>();
365         cardList.add(new Card("9", "H"));
366         cardList.add(new Card("7", "D"));
367         cardList.add(new Card("8", "C"));
368         cardList.add(new Card("11", "C"));
369         cardList.add(new Card("10", "S"));
370         CommunityCardSet cc = new CommunityCardSet(cardList);
371         hand1.add(new Card("14", "H"));
372         hand1.add(new Card("5", "D"));
373         hand2.add(new Card("14", "C"));
374         hand2.add(new Card("5", "S"));
375         StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
376         StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
377         int actual = studHand1.compareTo(studHand2);
378         Testing.assertEquals("test of tie when both have same ranks in
    stud hands between two hands", 0, actual);
379     }
380
381 }
382
383
```