

```
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 here
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    }
```

```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/Card.java
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
50 * converts into corresponding integer
51 * @param rank if rank is a string, fed into here
52 * @return integer indicating rank; returns 2 as default if string does not
53 * fit any of possible inputs
54 */
55 private int stringToInteger(String rank) {
56     if ((rank.equalsIgnoreCase("two") || rank.equalsIgnoreCase("2")){
57         return 2;
58     }
59     else if ((rank.equalsIgnoreCase("three") || rank.equalsIgnoreCase("3")){
60         return 3;
61     }
62     else if ((rank.equalsIgnoreCase("four") || rank.equalsIgnoreCase("4")){
63         return 4;
64     }
65     else if ((rank.equalsIgnoreCase("five") || rank.equalsIgnoreCase("5")){
66         return 5;
67     }
68     else if ((rank.equalsIgnoreCase("six") || rank.equalsIgnoreCase("6")){
69         return 6;
70     }
71     else if ((rank.equalsIgnoreCase("seven") || rank.equalsIgnoreCase("7")){
72         return 7;
73     }
74     else if ((rank.equalsIgnoreCase("eight") || rank.equalsIgnoreCase("8")){
75         return 8;
76     }
77     else if ((rank.equalsIgnoreCase("nine") || rank.equalsIgnoreCase("9")){
78         return 9;
79     }
80     return 2;
81 }
82
83 private String integerToString(int rank) {
84     switch (rank) {
85         case 2: return "two";
86         case 3: return "three";
87         case 4: return "four";
88         case 5: return "five";
89         case 6: return "six";
90         case 7: return "seven";
91         case 8: return "eight";
92         case 9: return "nine";
93         default: return "null";
94     }
95 }
96
97 public void printHand() {
98     for (int i = 0; i < hand.length; i++) {
99         System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
100    }
101 }
102
103 public void setHand(String[] cards) {
104     for (int i = 0; i < cards.length; i++) {
105         hand[i] = new Card(cards[i]);
106    }
107 }
108
109 public void shuffle() {
110     for (int i = 0; i < hand.length; i++) {
111         int j = random.nextInt(hand.length);
112         Card temp = hand[i];
113         hand[i] = hand[j];
114         hand[j] = temp;
115     }
116 }
117
118 public void dealHand() {
119     for (int i = 0; i < hand.length; i++) {
120         hand[i] = new Card(random.nextInt(13), random.nextInt(4));
121    }
122 }
123
124 public void printHand() {
125     for (int i = 0; i < hand.length; i++) {
126         System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
127    }
128 }
129
130 public void printHand(String suit) {
131     for (int i = 0; i < hand.length; i++) {
132         if (hand[i].getSuit().equalsIgnoreCase(suit)) {
133             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
134         }
135    }
136 }
137
138 public void printHand(int rank) {
139     for (int i = 0; i < hand.length; i++) {
140         if (hand[i].getRank() == rank) {
141             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
142         }
143    }
144 }
145
146 public void printHand(String suit, int rank) {
147     for (int i = 0; i < hand.length; i++) {
148         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
149             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
150         }
151    }
152 }
153
154 public void printHand(String suit, int rank, int count) {
155     for (int i = 0; i < hand.length; i++) {
156         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
157             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
158             count--;
159             if (count == 0) {
160                 break;
161             }
162         }
163    }
164 }
165
166 public void printHand(String suit, int rank, int count, int index) {
167     for (int i = index; i < hand.length; i++) {
168         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
169             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
170             count--;
171             if (count == 0) {
172                 break;
173             }
174         }
175    }
176 }
177
178 public void printHand(String suit, int rank, int count, int index, int value) {
179     for (int i = index; i < hand.length; i++) {
180         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
181             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
182             count--;
183             if (count == 0) {
184                 break;
185             }
186         }
187     }
188 }
189
190 public void printHand(String suit, int rank, int count, int index, int value, int value2) {
191     for (int i = index; i < hand.length; i++) {
192         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
193             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
194             count--;
195             if (count == 0) {
196                 break;
197             }
198         }
199     }
200 }
201
202 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3) {
203     for (int i = index; i < hand.length; i++) {
204         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
205             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
206             count--;
207             if (count == 0) {
208                 break;
209             }
210         }
211     }
212 }
213
214 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4) {
215     for (int i = index; i < hand.length; i++) {
216         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
217             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
218             count--;
219             if (count == 0) {
220                 break;
221             }
222         }
223     }
224 }
225
226 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5) {
227     for (int i = index; i < hand.length; i++) {
228         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
229             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
230             count--;
231             if (count == 0) {
232                 break;
233             }
234         }
235     }
236 }
237
238 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6) {
239     for (int i = index; i < hand.length; i++) {
240         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
241             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
242             count--;
243             if (count == 0) {
244                 break;
245             }
246         }
247     }
248 }
249
250 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7) {
251     for (int i = index; i < hand.length; i++) {
252         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
253             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
254             count--;
255             if (count == 0) {
256                 break;
257             }
258         }
259     }
260 }
261
262 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8) {
263     for (int i = index; i < hand.length; i++) {
264         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
265             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
266             count--;
267             if (count == 0) {
268                 break;
269             }
270         }
271     }
272 }
273
274 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9) {
275     for (int i = index; i < hand.length; i++) {
276         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
277             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
278             count--;
279             if (count == 0) {
280                 break;
281             }
282         }
283     }
284 }
285
286 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10) {
287     for (int i = index; i < hand.length; i++) {
288         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
289             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
290             count--;
291             if (count == 0) {
292                 break;
293             }
294         }
295     }
296 }
297
298 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10, int value11) {
299     for (int i = index; i < hand.length; i++) {
300         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
301             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
302             count--;
303             if (count == 0) {
304                 break;
305             }
306         }
307     }
308 }
309
310 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10, int value11, int value12) {
311     for (int i = index; i < hand.length; i++) {
312         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
313             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
314             count--;
315             if (count == 0) {
316                 break;
317             }
318         }
319     }
320 }
321
322 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10, int value11, int value12, int value13) {
323     for (int i = index; i < hand.length; i++) {
324         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
325             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
326             count--;
327             if (count == 0) {
328                 break;
329             }
330         }
331     }
332 }
333
334 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10, int value11, int value12, int value13, int value14) {
335     for (int i = index; i < hand.length; i++) {
336         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
337             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
338             count--;
339             if (count == 0) {
340                 break;
341             }
342         }
343     }
344 }
345
346 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10, int value11, int value12, int value13, int value14, int value15) {
347     for (int i = index; i < hand.length; i++) {
348         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
349             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
350             count--;
351             if (count == 0) {
352                 break;
353             }
354         }
355     }
356 }
357
358 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10, int value11, int value12, int value13, int value14, int value15, int value16) {
359     for (int i = index; i < hand.length; i++) {
360         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
361             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
362             count--;
363             if (count == 0) {
364                 break;
365             }
366         }
367     }
368 }
369
370 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10, int value11, int value12, int value13, int value14, int value15, int value16, int value17) {
371     for (int i = index; i < hand.length; i++) {
372         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
373             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
374             count--;
375             if (count == 0) {
376                 break;
377             }
378         }
379     }
380 }
381
382 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10, int value11, int value12, int value13, int value14, int value15, int value16, int value17, int value18) {
383     for (int i = index; i < hand.length; i++) {
384         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
385             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
386             count--;
387             if (count == 0) {
388                 break;
389             }
390         }
391     }
392 }
393
394 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10, int value11, int value12, int value13, int value14, int value15, int value16, int value17, int value18, int value19) {
395     for (int i = index; i < hand.length; i++) {
396         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
397             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
398             count--;
399             if (count == 0) {
400                 break;
401             }
402         }
403     }
404 }
405
406 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10, int value11, int value12, int value13, int value14, int value15, int value16, int value17, int value18, int value19, int value20) {
407     for (int i = index; i < hand.length; i++) {
408         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
409             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
410             count--;
411             if (count == 0) {
412                 break;
413             }
414         }
415     }
416 }
417
418 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10, int value11, int value12, int value13, int value14, int value15, int value16, int value17, int value18, int value19, int value20, int value21) {
419     for (int i = index; i < hand.length; i++) {
420         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
421             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
422             count--;
423             if (count == 0) {
424                 break;
425             }
426         }
427     }
428 }
429
430 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10, int value11, int value12, int value13, int value14, int value15, int value16, int value17, int value18, int value19, int value20, int value21, int value22) {
431     for (int i = index; i < hand.length; i++) {
432         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
433             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
434             count--;
435             if (count == 0) {
436                 break;
437             }
438         }
439     }
440 }
441
442 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10, int value11, int value12, int value13, int value14, int value15, int value16, int value17, int value18, int value19, int value20, int value21, int value22, int value23) {
443     for (int i = index; i < hand.length; i++) {
444         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
445             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
446             count--;
447             if (count == 0) {
448                 break;
449             }
450         }
451     }
452 }
453
454 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10, int value11, int value12, int value13, int value14, int value15, int value16, int value17, int value18, int value19, int value20, int value21, int value22, int value23, int value24) {
455     for (int i = index; i < hand.length; i++) {
456         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
457             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
458             count--;
459             if (count == 0) {
460                 break;
461             }
462         }
463     }
464 }
465
466 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10, int value11, int value12, int value13, int value14, int value15, int value16, int value17, int value18, int value19, int value20, int value21, int value22, int value23, int value24, int value25) {
467     for (int i = index; i < hand.length; i++) {
468         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
469             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
470             count--;
471             if (count == 0) {
472                 break;
473             }
474         }
475     }
476 }
477
478 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10, int value11, int value12, int value13, int value14, int value15, int value16, int value17, int value18, int value19, int value20, int value21, int value22, int value23, int value24, int value25, int value26) {
479     for (int i = index; i < hand.length; i++) {
480         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
481             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
482             count--;
483             if (count == 0) {
484                 break;
485             }
486         }
487     }
488 }
489
490 public void printHand(String suit, int rank, int count, int index, int value, int value2, int value3, int value4, int value5, int value6, int value7, int value8, int value9, int value10, int value11, int value12, int value13, int value14, int value15, int value16, int value17, int value18, int value19, int value20, int value21, int value22, int value23, int value24, int value25, int value26, int value27) {
491     for (int i = index; i < hand.length; i++) {
492         if (hand[i].getRank() == rank && hand[i].getSuit().equalsIgnoreCase(suit)) {
493             System.out.println(hand[i].getRank() + " of " + hand[i].getSuit());
494             count--;
495             if (count == 0) {
496                 break;
497             }
498         }
499     }
500 }
501
502 public void printHand(String suit, int rank,
```

```
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    /**
```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/Card.java

```
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 }
```

```
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", "
7     11", "12", "13", "14"};
8     private final String[] suits = {"Clubs", "Diamonds", "Hearts", "Spades"};
9     private static final int NUMBER_OF_CARDS=52;
10    private static final int NUMBER_OF_SUITS=4;
11    private static final int CARDS_IN_SUIT=13;
12    private static final int HAND_LENGTH = 5;
13    private ArrayList<Card> myDeck;
14    private int nextToDeal;
15
16    /**
17     * creates ArrayList to represent deck of cards
18     */
19    public Deck() {
20        nextToDeal = 0;
21        myDeck = new ArrayList<Card>();
22        for (String rank : ranks) {
23            for (String suit : suits) {
24                Card card = new Card(rank, suit);
25                myDeck.add(card);
26            }
27        }
28    }
29
30    /**
31     * returns shuffled deck by swapping card location with random card
32     * location and vice versa
33     */
34    public void shuffle() {
35        for (int oldCardIndex = nextToDeal; oldCardIndex < myDeck.size(); oldCardIndex++) {
36            int newCardIndex = ThreadLocalRandom.current().nextInt(myDeck.
37            size());
38            Card oldCard = myDeck.get(oldCardIndex);
39            Card newCard = myDeck.get(newCardIndex);
40            myDeck.set(oldCardIndex, newCard);
```

```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/Deck.java
39         myDeck.set(newCardIndex, oldCard);
40     }
41 }
42 /**
43 * returns size of deck by using nextToDeal as marker of which cards
44 have been dealt
45 */
46 public int size() {
47     return ((myDeck.size() - nextToDeal));
48 }
49 /**
50 * deals single card from created deck; progresses through deck by
51 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 * gathers deck to unshuffled state by setting nextToDeal to zero again (
64 moving divider between dealt and undealt
65 * cards to very front of deck
66 */
67 public void gather(){
68     nextToDeal = 0;
69 }
70 /**
71 * returns boolean indicating whether deck is empty or not
72 */
73 public boolean isEmpty(){
74     if (size() == 0){
75         return true;
76     }

```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/Deck.java

```
78     return false;
79 }
80
81 /**
82 * returns readable version of deck with each card in deck made into
83 string
84 */
85 public String toString(){
86     String toReturn = "";
87     for (int i = nextToDeal; i < myDeck.size(); i++){
88         Card card = myDeck.get(i);
89         toReturn = (toReturn + "|" + card.toString() + "|");
90     }
91     return toReturn;
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
5  * methods
6  * here are static so there's no need to construct a Testing object. Just call
7  * them
8  * with the class name like so:
9  * <p></p>
10 * <code>Testing.assertEquals("test description", expected, actual)</code>
11 *
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 {  
64     printTestCaseInfo(message, "" + expected, "" + actual);  
65     if (expected == actual) {  
66         pass();  
67     } else {  
68         fail(message);  
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 }
```

```
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 {
```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/Testing.java

```
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     }
167 }
```

```
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 }
```

```
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";
```

```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/CardTests.java
42     boolean actual = card.toString().equals(expectedCard);
43     Testing.assertEquals("Tests string card constructor with rank
44     spelled out input", true, actual);
45 }
46
47 public static void testingStringConstructorCase3() {
48     Card card = new Card("Four", "Hearts");
49     String expectedCard = "4 of Hearts";
50     boolean actual = card.toString().equals(expectedCard);
51     Testing.assertEquals("Tests string card constructor case
52     insensitivity", true, actual);
53 }
54
55 public static void testingStringConstructorFaceCard(){
56     Card card = new Card("13", "Diamonds");
57     String expectedCard = "King of Diamonds";
58     boolean actual = card.toString().equals(expectedCard);
59     Testing.assertEquals("Tests string card constructor with face card"
60     , true, actual);
61 }
62
63 public static void testingGetSuit(){
64     Card card = new Card("13", "Diamonds");
65     String suit = "Diamonds";
66     boolean actual = card.getSuit().equals(suit);
67     Testing.assertEquals("Tests string card constructor with face card"
68     , true, actual);
69 }
70
71 public static void testingGetRank(){
72     Card card = new Card("Four", "Hearts");
73     int rank = 4;
74     boolean actual = card.getRank() == (rank);
75     Testing.assertEquals("Tests string card constructor with face card"
76     , true, actual);
77 }
78

```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/CardTests.java

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
20 correctly", expected, actual);
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
29 unshuffled deck", expected, actual);
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
37 one card", expected, actual);
38    }
39}
```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/DeckTests.java

```
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
49      different amount of cards after both dealing one card", expected,
50      actual);
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
37     * new card only if poker hand has less than five cards
38     * @param card representation of playing card to add to hand
39     */
40    public void addCard(Card card) {
41        if (myHand.size() < HAND_LENGTH) {
42            myHand.add(card);
```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/PokerHand.java

```

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
59 array list
60 * @return list of ranks
61 */
62 private ArrayList<Integer> getRanks() {
63     for (int i = 0; i < HAND_LENGTH; i++) {
64         Card card = myHand.get(i);
65         Ranks.add(card.getRank());
66     }
67     return Ranks;
68 }
69
70 /**
71 * gets suits from all cards in poker hand and adds them to Suits array
72 list
73 * @return list of suits
74 */
75 private ArrayList<String> getSuits() {
76     for (int i = 0; i < HAND_LENGTH; i++) {
77         Card card = myHand.get(i);
78         Suits.add(card.getSuit());
79     }
80     return Suits;
81 }
82
83 /**
84 * gets the total value of the poker hand
85 * @return total value of hand
86 */
87 private int getTotalValue() {
88     int total = 0;
89     for (int i = 0; i < Ranks.size(); i++) {
90         total += Ranks.get(i);
91     }
92     return total;
93 }
94
95 /**
96 * gets the total number of suits in the poker hand
97 * @return total number of suits
98 */
99 private int getTotalSuits() {
100    int total = 0;
101    for (int i = 0; i < Suits.size(); i++) {
102        total += Suits.get(i);
103    }
104    return total;
105 }
106
107 /**
108 * gets the total number of ranks in the poker hand
109 * @return total number of ranks
110 */
111 private int getTotalRanks() {
112    int total = 0;
113    for (int i = 0; i < Ranks.size(); i++) {
114        total += Ranks.get(i);
115    }
116    return total;
117 }
118
119 /**
120 * gets the total number of cards in the poker hand
121 * @return total number of cards
122 */
123 private int getTotalCards() {
124    int total = 0;
125    for (int i = 0; i < myHand.size(); i++) {
126        total += myHand.get(i);
127    }
128    return total;
129 }
130
131 /**
132 * gets the total value of the poker hand
133 * @return total value of hand
134 */
135 private int getTotalValue() {
136     int total = 0;
137     for (int i = 0; i < Ranks.size(); i++) {
138         total += Ranks.get(i);
139     }
140     return total;
141 }
142
143 /**
144 * gets the total number of suits in the poker hand
145 * @return total number of suits
146 */
147 private int getTotalSuits() {
148    int total = 0;
149    for (int i = 0; i < Suits.size(); i++) {
150        total += Suits.get(i);
151    }
152    return total;
153 }
154
155 /**
156 * gets the total number of ranks in the poker hand
157 * @return total number of ranks
158 */
159 private int getTotalRanks() {
160    int total = 0;
161    for (int i = 0; i < Ranks.size(); i++) {
162        total += Ranks.get(i);
163    }
164    return total;
165 }
166
167 /**
168 * gets the total number of cards in the poker hand
169 * @return total number of cards
170 */
171 private int getTotalCards() {
172    int total = 0;
173    for (int i = 0; i < myHand.size(); i++) {
174        total += myHand.get(i);
175    }
176    return total;
177 }
178
179 /**
180 * gets the total value of the poker hand
181 * @return total value of hand
182 */
183 private int getTotalValue() {
184     int total = 0;
185     for (int i = 0; i < Ranks.size(); i++) {
186         total += Ranks.get(i);
187     }
188     return total;
189 }
190
191 /**
192 * gets the total number of suits in the poker hand
193 * @return total number of suits
194 */
195 private int getTotalSuits() {
196    int total = 0;
197    for (int i = 0; i < Suits.size(); i++) {
198        total += Suits.get(i);
199    }
200    return total;
201 }
202
203 /**
204 * gets the total number of ranks in the poker hand
205 * @return total number of ranks
206 */
207 private int getTotalRanks() {
208    int total = 0;
209    for (int i = 0; i < Ranks.size(); i++) {
210        total += Ranks.get(i);
211    }
212    return total;
213 }
214
215 /**
216 * gets the total number of cards in the poker hand
217 * @return total number of cards
218 */
219 private int getTotalCards() {
220    int total = 0;
221    for (int i = 0; i < myHand.size(); i++) {
222        total += myHand.get(i);
223    }
224    return total;
225 }
226
227 /**
228 * gets the total value of the poker hand
229 * @return total value of hand
230 */
231 private int getTotalValue() {
232     int total = 0;
233     for (int i = 0; i < Ranks.size(); i++) {
234         total += Ranks.get(i);
235     }
236     return total;
237 }
238
239 /**
240 * gets the total number of suits in the poker hand
241 * @return total number of suits
242 */
243 private int getTotalSuits() {
244    int total = 0;
245    for (int i = 0; i < Suits.size(); i++) {
246        total += Suits.get(i);
247    }
248    return total;
249 }
250
251 /**
252 * gets the total number of ranks in the poker hand
253 * @return total number of ranks
254 */
255 private int getTotalRanks() {
256    int total = 0;
257    for (int i = 0; i < Ranks.size(); i++) {
258        total += Ranks.get(i);
259    }
260    return total;
261 }
262
263 /**
264 * gets the total number of cards in the poker hand
265 * @return total number of cards
266 */
267 private int getTotalCards() {
268    int total = 0;
269    for (int i = 0; i < myHand.size(); i++) {
270        total += myHand.get(i);
271    }
272    return total;
273 }
274
275 /**
276 * gets the total value of the poker hand
277 * @return total value of hand
278 */
279 private int getTotalValue() {
280     int total = 0;
281     for (int i = 0; i < Ranks.size(); i++) {
282         total += Ranks.get(i);
283     }
284     return total;
285 }
286
287 /**
288 * gets the total number of suits in the poker hand
289 * @return total number of suits
290 */
291 private int getTotalSuits() {
292    int total = 0;
293    for (int i = 0; i < Suits.size(); i++) {
294        total += Suits.get(i);
295    }
296    return total;
297 }
298
299 /**
300 * gets the total number of ranks in the poker hand
301 * @return total number of ranks
302 */
303 private int getTotalRanks() {
304    int total = 0;
305    for (int i = 0; i < Ranks.size(); i++) {
306        total += Ranks.get(i);
307    }
308    return total;
309 }
310
311 /**
312 * gets the total number of cards in the poker hand
313 * @return total number of cards
314 */
315 private int getTotalCards() {
316    int total = 0;
317    for (int i = 0; i < myHand.size(); i++) {
318        total += myHand.get(i);
319    }
320    return total;
321 }
322
323 /**
324 * gets the total value of the poker hand
325 * @return total value of hand
326 */
327 private int getTotalValue() {
328     int total = 0;
329     for (int i = 0; i < Ranks.size(); i++) {
330         total += Ranks.get(i);
331     }
332     return total;
333 }
334
335 /**
336 * gets the total number of suits in the poker hand
337 * @return total number of suits
338 */
339 private int getTotalSuits() {
340    int total = 0;
341    for (int i = 0; i < Suits.size(); i++) {
342        total += Suits.get(i);
343    }
344    return total;
345 }
346
347 /**
348 * gets the total number of ranks in the poker hand
349 * @return total number of ranks
350 */
351 private int getTotalRanks() {
352    int total = 0;
353    for (int i = 0; i < Ranks.size(); i++) {
354        total += Ranks.get(i);
355    }
356    return total;
357 }
358
359 /**
360 * gets the total number of cards in the poker hand
361 * @return total number of cards
362 */
363 private int getTotalCards() {
364    int total = 0;
365    for (int i = 0; i < myHand.size(); i++) {
366        total += myHand.get(i);
367    }
368    return total;
369 }
370
371 /**
372 * gets the total value of the poker hand
373 * @return total value of hand
374 */
375 private int getTotalValue() {
376     int total = 0;
377     for (int i = 0; i < Ranks.size(); i++) {
378         total += Ranks.get(i);
379     }
380     return total;
381 }
382
383 /**
384 * gets the total number of suits in the poker hand
385 * @return total number of suits
386 */
387 private int getTotalSuits() {
388    int total = 0;
389    for (int i = 0; i < Suits.size(); i++) {
390        total += Suits.get(i);
391    }
392    return total;
393 }
394
395 /**
396 * gets the total number of ranks in the poker hand
397 * @return total number of ranks
398 */
399 private int getTotalRanks() {
400    int total = 0;
401    for (int i = 0; i < Ranks.size(); i++) {
402        total += Ranks.get(i);
403    }
404    return total;
405 }
406
407 /**
408 * gets the total number of cards in the poker hand
409 * @return total number of cards
410 */
411 private int getTotalCards() {
412    int total = 0;
413    for (int i = 0; i < myHand.size(); i++) {
414        total += myHand.get(i);
415    }
416    return total;
417 }
418
419 /**
420 * gets the total value of the poker hand
421 * @return total value of hand
422 */
423 private int getTotalValue() {
424     int total = 0;
425     for (int i = 0; i < Ranks.size(); i++) {
426         total += Ranks.get(i);
427     }
428     return total;
429 }
430
431 /**
432 * gets the total number of suits in the poker hand
433 * @return total number of suits
434 */
435 private int getTotalSuits() {
436    int total = 0;
437    for (int i = 0; i < Suits.size(); i++) {
438        total += Suits.get(i);
439    }
440    return total;
441 }
442
443 /**
444 * gets the total number of ranks in the poker hand
445 * @return total number of ranks
446 */
447 private int getTotalRanks() {
448    int total = 0;
449    for (int i = 0; i < Ranks.size(); i++) {
450        total += Ranks.get(i);
451    }
452    return total;
453 }
454
455 /**
456 * gets the total number of cards in the poker hand
457 * @return total number of cards
458 */
459 private int getTotalCards() {
460    int total = 0;
461    for (int i = 0; i < myHand.size(); i++) {
462        total += myHand.get(i);
463    }
464    return total;
465 }
466
467 /**
468 * gets the total value of the poker hand
469 * @return total value of hand
470 */
471 private int getTotalValue() {
472     int total = 0;
473     for (int i = 0; i < Ranks.size(); i++) {
474         total += Ranks.get(i);
475     }
476     return total;
477 }
478
479 /**
480 * gets the total number of suits in the poker hand
481 * @return total number of suits
482 */
483 private int getTotalSuits() {
484    int total = 0;
485    for (int i = 0; i < Suits.size(); i++) {
486        total += Suits.get(i);
487    }
488    return total;
489 }
490
491 /**
492 * gets the total number of ranks in the poker hand
493 * @return total number of ranks
494 */
495 private int getTotalRanks() {
496    int total = 0;
497    for (int i = 0; i < Ranks.size(); i++) {
498        total += Ranks.get(i);
499    }
500    return total;
501 }
502
503 /**
504 * gets the total number of cards in the poker hand
505 * @return total number of cards
506 */
507 private int getTotalCards() {
508    int total = 0;
509    for (int i = 0; i < myHand.size(); i++) {
510        total += myHand.get(i);
511    }
512    return total;
513 }
514
515 /**
516 * gets the total value of the poker hand
517 * @return total value of hand
518 */
519 private int getTotalValue() {
520     int total = 0;
521     for (int i = 0; i < Ranks.size(); i++) {
522         total += Ranks.get(i);
523     }
524     return total;
525 }
526
527 /**
528 * gets the total number of suits in the poker hand
529 * @return total number of suits
530 */
531 private int getTotalSuits() {
532    int total = 0;
533    for (int i = 0; i < Suits.size(); i++) {
534        total += Suits.get(i);
535    }
536    return total;
537 }
538
539 /**
540 * gets the total number of ranks in the poker hand
541 * @return total number of ranks
542 */
543 private int getTotalRanks() {
544    int total = 0;
545    for (int i = 0; i < Ranks.size(); i++) {
546        total += Ranks.get(i);
547    }
548    return total;
549 }
550
551 /**
552 * gets the total number of cards in the poker hand
553 * @return total number of cards
554 */
555 private int getTotalCards() {
556    int total = 0;
557    for (int i = 0; i < myHand.size(); i++) {
558        total += myHand.get(i);
559    }
560    return total;
561 }
562
563 /**
564 * gets the total value of the poker hand
565 * @return total value of hand
566 */
567 private int getTotalValue() {
568     int total = 0;
569     for (int i = 0; i < Ranks.size(); i++) {
570         total += Ranks.get(i);
571     }
572     return total;
573 }
574
575 /**
576 * gets the total number of suits in the poker hand
577 * @return total number of suits
578 */
579 private int getTotalSuits() {
580    int total = 0;
581    for (int i = 0; i < Suits.size(); i++) {
582        total += Suits.get(i);
583    }
584    return total;
585 }
586
587 /**
588 * gets the total number of ranks in the poker hand
589 * @return total number of ranks
590 */
591 private int getTotalRanks() {
592    int total = 0;
593    for (int i = 0; i < Ranks.size(); i++) {
594        total += Ranks.get(i);
595    }
596    return total;
597 }
598
599 /**
600 * gets the total number of cards in the poker hand
601 * @return total number of cards
602 */
603 private int getTotalCards() {
604    int total = 0;
605    for (int i = 0; i < myHand.size(); i++) {
606        total += myHand.get(i);
607    }
608    return total;
609 }
610
611 /**
612 * gets the total value of the poker hand
613 * @return total value of hand
614 */
615 private int getTotalValue() {
616     int total = 0;
617     for (int i = 0; i < Ranks.size(); i++) {
618         total += Ranks.get(i);
619     }
620     return total;
621 }
622
623 /**
624 * gets the total number of suits in the poker hand
625 * @return total number of suits
626 */
627 private int getTotalSuits() {
628    int total = 0;
629    for (int i = 0; i < Suits.size(); i++) {
630        total += Suits.get(i);
631    }
632    return total;
633 }
634
635 /**
636 * gets the total number of ranks in the poker hand
637 * @return total number of ranks
638 */
639 private int getTotalRanks() {
640    int total = 0;
641    for (int i = 0; i < Ranks.size(); i++) {
642        total += Ranks.get(i);
643    }
644    return total;
645 }
646
647 /**
648 * gets the total number of cards in the poker hand
649 * @return total number of cards
650 */
651 private int getTotalCards() {
652    int total = 0;
653    for (int i = 0; i < myHand.size(); i++) {
654        total += myHand.get(i);
655    }
656    return total;
657 }
658
659 /**
660 * gets the total value of the poker hand
661 * @return total value of hand
662 */
663 private int getTotalValue() {
664     int total = 0;
665     for (int i = 0; i < Ranks.size(); i++) {
666         total += Ranks.get(i);
667     }
668     return total;
669 }
670
671 /**
672 * gets the total number of suits in the poker hand
673 * @return total number of suits
674 */
675 private int getTotalSuits() {
676    int total = 0;
677    for (int i = 0; i < Suits.size(); i++) {
678        total += Suits.get(i);
679    }
680    return total;
681 }
682
683 /**
684 * gets the total number of ranks in the poker hand
685 * @return total number of ranks
686 */
687 private int getTotalRanks() {
688    int total = 0;
689    for (int i = 0; i < Ranks.size(); i++) {
690        total += Ranks.get(i);
691    }
692    return total;
693 }
694
695 /**
696 * gets the total number of cards in the poker hand
697 * @return total number of cards
698 */
699 private int getTotalCards() {
700    int total = 0;
701    for (int i = 0; i < myHand.size(); i++) {
702        total += myHand.get(i);
703    }
704    return total;
705 }
706
707 /**
708 * gets the total value of the poker hand
709 * @return total value of hand
710 */
711 private int getTotalValue() {
712     int total = 0;
713     for (int i = 0; i < Ranks.size(); i++) {
714         total += Ranks.get(i);
715     }
716     return total;
717 }
718
719 /**
720 * gets the total number of suits in the poker hand
721 * @return total number of suits
722 */
723 private int getTotalSuits() {
724    int total = 0;
725    for (int i = 0; i < Suits.size(); i++) {
726        total += Suits.get(i);
727    }
728    return total;
729 }
730
731 /**
732 * gets the total number of ranks in the poker hand
733 * @return total number of ranks
734 */
735 private int getTotalRanks() {
736    int total = 0;
737    for (int i = 0; i < Ranks.size(); i++) {
738        total += Ranks.get(i);
739    }
740    return total;
741 }
742
743 /**
744 * gets the total number of cards in the poker hand
745 * @return total number of cards
746 */
747 private int getTotalCards() {
748    int total = 0;
749    for (int i = 0; i < myHand.size(); i++) {
750        total += myHand.get(i);
751    }
752    return total;
753 }
754
755 /**
756 * gets the total value of the poker hand
757 * @return total value of hand
758 */
759 private int getTotalValue() {
760     int total = 0;
761     for (int i = 0; i < Ranks.size(); i++) {
762         total += Ranks.get(i);
763     }
764     return total;
765 }
766
767 /**
768 * gets the total number of suits in the poker hand
769 * @return total number of suits
770 */
771 private int getTotalSuits() {
772    int total = 0;
773    for (int i = 0; i < Suits.size(); i++) {
774        total += Suits.get(i);
775    }
776    return total;
777 }
778
779 /**
780 * gets the total number of ranks in the poker hand
781 * @return total number of ranks
782 */
783 private int getTotalRanks() {
784    int total = 0;
785    for (int i = 0; i < Ranks.size(); i++) {
786        total += Ranks.get(i);
787    }
788    return total;
789 }
790
791 /**
792 * gets the total number of cards in the poker hand
793 * @return total number of cards
794 */
795 private int getTotalCards() {
796    int total = 0;
797    for (int i = 0; i < myHand.size(); i++) {
798        total += myHand.get(i);
799    }
800    return total;
801 }
802
803 /**
804 * gets the total value of the poker hand
805 * @return total value of hand
806 */
807 private int getTotalValue() {
808     int total = 0;
809     for (int i = 0; i < Ranks.size(); i++) {
810         total += Ranks.get(i);
811     }
812     return total;
813 }
814
815 /**
816 * gets the total number of suits in the poker hand
817 * @return total number of suits
818 */
819 private int getTotalSuits() {
820    int total = 0;
821    for (int i = 0; i < Suits.size(); i++) {
822        total += Suits.get(i);
823    }
824    return total;
825 }
826
827 /**
828 * gets the total number of ranks in the poker hand
829 * @return total number of ranks
830 */
831 private int getTotalRanks() {
832    int total = 0;
833    for (int i = 0; i < Ranks.size(); i++) {
834        total += Ranks.get(i);
835    }
836    return total;
837 }
838
839 /**
840 * gets the total number of cards in the poker hand
841 * @return total number of cards
842 */
843 private int getTotalCards() {
844    int total = 0;
845    for (int i = 0; i < myHand.size(); i++) {
846        total += myHand.get(i);
847    }
848    return total;
849 }
850
851 /**
852 * gets the total value of the poker hand
853 * @return total value of hand
854 */
855 private int getTotalValue() {
856     int total = 0;
857     for (int i = 0; i < Ranks.size(); i++) {
858         total += Ranks.get(i);
859     }
860     return total;
861 }
862
863 /**
864 * gets the total number of suits in the poker hand
865 * @return total number of suits
866 */
867 private int getTotalSuits() {
868    int total = 0;
869    for (int i = 0; i < Suits.size(); i++) {
870        total += Suits.get(i);
871    }
872    return total;
873 }
874
875 /**
876 * gets the total number of ranks in the poker hand
877 * @return total number of ranks
878 */
879 private int getTotalRanks() {
880    int total = 0;
881    for (int i = 0; i < Ranks.size(); i++) {
882        total += Ranks.get(i);
883    }
884    return total;
885 }
886
887 /**
888 * gets the total number of cards in the poker hand
889 * @return total number of cards
890 */
891 private int getTotalCards() {
892    int total = 0;
893    for (int i = 0; i < myHand.size(); i++) {
894        total += myHand.get(i);
895    }
896    return total;
897 }
898
899 /**
900 * gets the total value of the poker hand
901 * @return total value of hand
902 */
903 private int getTotalValue() {
904     int total = 0;
905     for (int i = 0; i < Ranks.size(); i++) {
906         total += Ranks.get(i);
907     }
908     return total;
909 }
910
911 /**
912 * gets the total number of suits in the poker hand
913 * @return total number of suits
914 */
915 private int getTotalSuits() {
916    int total = 0;
917    for (int i = 0; i < Suits.size(); i++) {
918        total += Suits.get(i);
919    }
920    return total;
921 }
922
923 /**
924 * gets the total number of ranks in the poker hand
925 * @return total number of ranks
926 */
927 private int getTotalRanks() {
928    int total = 0;
929    for (int i = 0; i < Ranks.size(); i++) {
930        total += Ranks.get(i);
931    }
932    return total;
933 }
934
935 /**
936 * gets the total number of cards in the poker hand
937 * @return total number of cards
938 */
939 private int getTotalCards() {
940    int total = 0;
941    for (int i = 0; i < myHand.size(); i++) {
942        total += myHand.get(i);
943    }
944    return total;
945 }
946
947 /**
948 * gets the total value of the poker hand
949 * @return total value of hand
950 */
951 private int getTotalValue() {
952     int total = 0;
953     for (int i = 0; i < Ranks.size(); i++) {
954         total += Ranks.get(i);
955     }
956     return total;
957 }
958
959 /**
960 * gets the total number of suits in the poker hand
961 * @return total number of suits
962 */
963 private int getTotalSuits() {
964    int total = 0;
965    for (int i = 0; i < Suits.size(); i++) {
966        total += Suits.get(i);
967    }
968    return total;
969 }
970
971 /**
972 * gets the total number of ranks in the poker hand
973 * @return total number of ranks
974 */
975 private int getTotalRanks() {
976    int total = 0;
977    for (int i = 0; i < Ranks.size(); i++) {
978        total += Ranks.get(i);
979    }
980    return total;
981 }
982
983 /**
984 * gets the total number of cards in the poker hand
985 * @return total number of cards
986 */
987 private int getTotalCards() {
988    int total = 0;
989    for (int i = 0; i < myHand.size(); i++) {
990        total += myHand.get(i);
991    }
992    return total;
993 }
994
995 /**
996 * gets the total value of the poker hand
997 * @return total value of hand
998 */
999 private int getTotalValue() {
1000    int total = 0;
1001    for (int i = 0; i < Ranks.size(); i++) {
1002        total += Ranks.get(i);
1003    }
1004    return total;
1005 }
1006
1007 /**
1008 * gets the total number of suits in the poker hand
1009 * @return total number of suits
1010 */
1011 private int getTotalSuits() {
1012    int total = 0;
1013    for (int i = 0; i < Suits.size(); i++) {
1014        total += Suits.get(i);
1015    }
1016    return total;
1017 }
1018
1019 /**
1020 * gets the total number of ranks in the poker hand
1021 * @return total number of ranks
1022 */
1023 private int getTotalRanks() {
1024    int total = 0;
1025    for (int i = 0; i < Ranks.size(); i++) {
1026        total += Ranks.get(i);
1027    }
1028    return total;
1029 }
1030
1031 /**
1032 * gets the total number of cards in the poker hand
1033 * @return total number of cards
1034 */
1035 private int getTotalCards() {
1036    int total = 0;
1037    for (int i = 0; i < myHand.size(); i++) {
1038        total += myHand.get(i);
1039    }
1040    return total;
1041 }
1042
1043 /**
1044 * gets the total value of the poker hand
1045 * @return total value of hand
1046 */
1047 private int getTotalValue() {
1048     int total = 0;
1049     for (int i = 0; i < Ranks.size(); i++) {
1050         total += Ranks.get(i);
1051     }
1052     return total;
1053 }
1054
1055 /**
1056 * gets the total number of suits in the poker hand
1057 * @return total number of suits
1058 */
1059 private int getTotalSuits() {
1060    int total = 0;
1061    for (int i = 0; i < Suits.size(); i++) {
1062        total += Suits.get(i);
1063    }
1064    return total;
1065 }
1066
1067 /**
1068 * gets the total number of ranks in the poker hand
1069 * @return total number of ranks
1070 */
1071 private int getTotalRanks() {
1072    int total = 0;
1073    for (int i = 0; i < Ranks.size(); i++) {
1074        total += Ranks.get(i);
1075    }
1076    return total;
1077 }
1078
1079 /**
1080 * gets the total number of cards in the poker hand
1081 * @return total number of cards
1082 */
1083 private int getTotalCards() {
1084    int total = 0;
1085    for (int i = 0; i < myHand.size(); i++) {
1086        total += myHand.get(i);
1087    }
1088    return total;
1089 }
1090
1091 /**
1092 * gets the total value of the poker hand
1093 * @return total value of hand
1094 */
1095 private int getTotalValue() {
1096     int total = 0;
1097     for (int i = 0; i < Ranks.size(); i++) {
1098         total += Ranks.get(i);
1099     }
1100     return total;
1101 }
1102
1103 /**
1104 * gets the total number of suits in the poker hand
1105 * @return total number of suits
1106 */
1107 private int getTotalSuits() {
1108    int total = 0;
1109    for (int i = 0; i < Suits.size(); i++) {
1110        total += Suits.get(i);
1111    }
1112    return total;
1113 }
1114
1115 /**
1116 * gets the total number of ranks in the poker hand
1117 * @return total number of ranks
1118 */
1119 private int getTotalRanks() {
1120    int total = 0;
1121    for (int i = 0; i < Ranks.size(); i++) {
1122        total += Ranks.get(i);
1123    }
1124    return total;
1125 }
1126
1127 /**
1128 * gets the total number of cards in the poker hand
1129 * @return total number of cards
1130 */
1131 private int getTotalCards() {
1132    int total = 0;
1133    for (int i = 0; i < myHand.size(); i++) {
1134        total += myHand.get(i);
1135    }
1136    return total;
1137 }
1138
1139 /**
1140 * gets the total value of the poker hand
1141 * @return total value of hand
1142 */
1143 private int getTotalValue() {
1144     int total = 0;
1145     for (int i = 0; i < Ranks.size(); i++) {
1146         total += Ranks.get(i);
1147     }
1148     return total;
1149 }
1150
1151 /**
1152 * gets the total number of suits in the poker hand
1153 * @return total number of suits
1154 */
1155 private int getTotalSuits() {
1156    int total = 0;
1157    for (int i = 0; i < Suits.size(); i++) {
1158        total += Suits.get(i);
1159    }
1160    return total;
1161 }
1162
1163 /**
1164 * gets the total number of ranks in the poker hand
1165 * @return total number of ranks
1166 */
1167 private int getTotalRanks() {
1168    int total = 0;
1169    for (int i = 0; i < Ranks.size(); i++) {
1170        total += Ranks.get(i);
1171    }
1172    return total;
1173 }
1174
1175 /**
1176 * gets the total number of cards in the poker hand
1177 * @return total number of cards
1178 */
1179 private int getTotalCards() {
1180    int total = 0;
1181    for (int i = 0; i < myHand.size(); i++) {
1182        total += myHand.get(i);
1183    }
1184    return total;
1185 }
1186
1187 /**
1188 * gets the total value of the poker hand
1189 * @return total value of hand
1190 */
1191 private int getTotalValue() {
1192     int total = 0;
1193     for (int i = 0; i < Ranks.size(); i++) {
1194         total += Ranks.get(i);
1195     }
1196     return total;
1197 }
1198
1199 /**
1200 * gets the total number of suits in the poker hand
1201 * @return total number of suits
1202 */
1203 private int getTotalSuits() {
1204    int total = 0;
1205    for (int i = 0; i < Suits.size(); i++) {
1206        total += Suits.get(i);
1207    }
1208    return total;
1209 }
1210
1211 /**
1212 * gets the total number of ranks in the poker hand
1213 * @return total number of ranks
1214 */
1215 private int getTotalRanks() {
1216    int total = 0;
1217    for (int i = 0; i < Ranks.size(); i++) {
1218        total += Ranks.get(i);
1219    }
1220    return total;
1221 }
1222
1223 /**
1224 * gets the total number of cards in the poker hand
1225 * @return total number of cards
1226 */
1227 private int getTotalCards() {
1228    int total = 0;
1229    for (int i = 0; i < myHand.size(); i++) {
1230        total += myHand.get(i);
1231    }
1232    return total;
1233 }
1234
1235 /**
1236 * gets the total value of the poker hand
1237 * @return total value of hand
1238 */
1239 private int getTotalValue() {
1240     int total = 0;
1241     for (int i = 0; i < Ranks.size(); i++) {
1242         total += Ranks.get(i);
1243     }
1244     return total;
1245 }
1246
1247 /**
1248 * gets the total number of suits in the poker hand
1249 * @return total number of suits
1250 */
1251 private int getTotalSuits() {
1252    int total = 0;
1253    for (int i = 0; i < Suits.size(); i++) {
1254        total += Suits.get(i);
1255    }
1256    return total;
1257 }
1258
1259 /**
1260 * gets the total number of ranks in the poker hand
1261 * @return total number of ranks
1262 */
1263 private int getTotalRanks() {
1264    int total = 0;
1265    for (int i = 0; i < Ranks.size(); i++) {
1266        total += Ranks.get(i);
1267    }
1268    return total;
1269 }
1270
1271 /**
1272 * gets the total number of cards in the poker hand
1273 * @return total number of cards
1274 */
1275 private int getTotalCards() {
1276    int total = 0;
1277    for (int i = 0; i < myHand.size(); i++) {
1278        total += myHand.get(i);
1279    }
1280    return total;
1281 }
1282
1283 /**
1284 * gets the total value of the poker hand
1285 * @return total value of hand
1286 */
1287 private int getTotalValue() {
1288     int total = 0;
1289     for (int i = 0; i < Ranks.size(); i++) {
1290         total += Ranks.get(i);
1291     }
1292     return total;
1293 }
1294
1295 /**
1296 * gets the total number of suits in the poker hand
1297 * @return total number of suits
1298 */
1299 private int getTotalSuits() {
1300    int total = 0;
1301    for (int i = 0; i < Suits.size(); i++) {
1302        total += Suits.get(i);
1303    }
1304    return total;
1305 }
1306
1307 /**
1308 * gets the total number of ranks in the poker hand
1309 * @return total number of ranks
1310 */
1311 private int getTotalRanks() {
1312    int total = 0;
1313    for (int i = 0; i < Ranks.size(); i++) {
1314        total += Ranks.get(i);
1315    }
1316    return total;
1317 }
1318
1319 /**
1320 * gets the total number of cards in the poker hand
1321 * @return total number of cards
1322 */
1323 private int getTotalCards() {
1324    int total = 0;
1325    for (int i = 0; i < myHand.size(); i++) {
1326        total += myHand.get(i);
1327    }
1328    return total;
1329 }
1330
1331 /**
1332 * gets the total value of the poker hand
1333 * @return total value of hand
1334 */
1335 private int getTotal
```

```
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
103     * have a pair;
104     * if pair is found, rank of pair is found and removed twice from ranks
105     * list to get rid of pair;
106     * pair rank is also saved to pair list once to see what rank pair was
107     * @return modified list of ranks once pairs have been removed
108     */
109     private ArrayList<Integer> checkIfPair() {
110         for (int i = 0; i < Ranks.size(); i++) {
111             for (int j = i + 1; j < Ranks.size(); j++) {
112                 if (Ranks.get(i) == Ranks.get(j)) {
113                     int pairRank = Ranks.get(i);
114                     Pairs.add(Ranks.get(i));
115                     Ranks.remove(Integer.valueOf(pairRank));
116                     Ranks.remove(Integer.valueOf(pairRank));
117                     return Ranks;
118                 }
119             }
120         }
121     }
```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/PokerHand.java

```

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
132     * pairs hand has; runs checkIfPair() again if
133     * length is 3 to check for two pairs
134     * @return string twoPair if size of ranks is 1, meaning two sets of pairs
135     * were removed, string onePair if size of
136     * ranks is 3, and null if size is neither 1 or 3
137     */
138    public String getPairType() {
139        getRanks();
140        checkIfPair();
141        if (Ranks.size() == 3) {
142            checkIfPair();
143            if (Ranks.size() == 1) {
144                Ranks.clear();
145                return "twoPair";
146            }
147            if (Ranks.size() == 3) {
148                Ranks.clear();
149                return "onePair";
150            }
151        }
152
153        /**
154         * sorts ranks in descending order
155         * @return first rank after sorting, meaning it returns the highest value
156         * rank in list
157         */
158        private int getHighestRank() {
159            Collections.sort(Ranks, Collections.reverseOrder());
160            return Ranks.get(0);
161        }

```

```

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
173  * highest rank value from remaining ranks;
174  * if there is more than one remaining rank, checkIfPair() is run again to
175  * check for two pairs; if remaining ranks
176  * still had more than one rank, ranks are sorted in descending order;
177  * first rank from each list of remaining ranks
178  * is compared
179  * @param other is hand to compare this hand to
180  * @return a negative number if self is worth LESS than other,
181  * zero if they are worth the SAME (a tie), and a positive number if
182  * self is worth MORE than other
183  */
184 private int getRemainingRankWinner(PokerHand other){
185   getRanks();
186   other.getRanks();
187   ArrayList<Integer> RemainingRanks1 = checkIfPair();
188   ArrayList<Integer> RemainingRanks2 = other.checkIfPair();
189   if (RemainingRanks1.size() != 1){
190     checkIfPair();
191     other.checkIfPair();
192     if (RemainingRanks1.size() != 1){
193       Collections.sort(RemainingRanks1, Collections.<Integer>
194       reverseOrder());
195       Collections.sort(RemainingRanks2, Collections.<Integer>
196       reverseOrder());
197     }
198   }
199   for (int i = 0; i < RemainingRanks1.size(); i++){
200     if (RemainingRanks1.get(i) > RemainingRanks2.get(i)){
201       Ranks.clear();
202       other.Ranks.clear();
203     }
204   }
205 }
```

```

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,
211 gets which set of pairs has highest rank;
212 * if both sets of pairs share the same ranks, getRemainingWinner() is
213 run to determine which of remaining non-pair
214 * ranks have highest value
215 * param other is hand to compare this hand to
216 * return a negative number if self is worth LESS than other,
217 * zero if they are worth the SAME (a tie), and a positive number if
218 * self is worth MORE than other
219 */
220 private int getPairsWinner(PokerHand other) {
221     ArrayList<Integer> pairs = getPairs();
222     ArrayList<Integer> otherPairs = other.getPairs();
223     ArrayList<Integer> Pairs1 = new ArrayList<Integer>();
224     ArrayList<Integer> Pairs2 = new ArrayList<Integer>();
225     Pairs1.add(pairs.get(0));
226     Pairs1.add(pairs.get(1));
227     Pairs2.add(otherPairs.get(0));
228     Pairs2.add(otherPairs.get(1));
229     Collections.sort(pairs, Collections.<Integer>reverseOrder());
230     Collections.sort(otherPairs, Collections.<Integer>reverseOrder());
231     for (int i = 0; i < MAX_PAIRS; i++) {
232         if (Pairs1.get(i) > Pairs2.get(i)) {
233             return 1;
234         }
235         if (Pairs1.get(i) < Pairs2.get(i)) {
236             return -1;
237         }
238     }
239     return getRemainingRankWinner(other);

```

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

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/PokerHand.java

```

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         return 0;
298     }
299 }
300 /**
301 * helper method for compareTo; gets results from isFlush method and
302 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())) {

```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/PokerHand.java

```

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

```

```

351     * helper method for compareTo; gets results from high card winner
352     * method
353     * @param other the hand to compare this hand to
354     * @return a negative number if this is worth LESS than other, zero
355     * if they are worth the SAME, and a positive number if this is worth
356     * MORE than other
357     */
358     private int compareToHighCard(PokerHand other){
359         if (getHighCardWinner(other) == 1) {
360             return 1;
361         }
362         if (getHighCardWinner(other) == -1) {
363             return -1;
364         }
365         return 0;
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     * returns readable version of hand where all cards are converted to
386     * strings
387     */
388     public String toString(){
389         String toReturn = "";
390         for (int i = 0; i < HAND_LENGTH; i++){

```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/PokerHand.java

```
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
14     * card hand, a stud poker hand, and a master hand
15     * which contains both community cards and stud cards
16     * @param cc community cards
17     * @param cardList list of card objects to add to stud poker hand
18     */
19     public StudPokerHand(CommunityCardSet cc, ArrayList<Card> cardList){
20         for (int i = 0; i < HAND_LENGTH; i++){
21             communityCards.add(cc.getIthCard(i));
22             masterHand.add(cc.getIthCard(i));
23         }
24         for (int j = 0; j < STUD_CARDS_LENGTH; j++){
25             studHand.add(cardList.get(j));
26             masterHand.add(cardList.get(j));
27         }
28     }
29     /**
30     * constructor for stud poker hand; makes three hands; a community
31     * card hand, a stud poker hand, and a master hand
32     * which contains both community cards and stud cards
33     * @param cc community cards
34     * @param gameDeck deck of card objects to take two card objects to
35     * add to stud poker hand
36     */
37     public StudPokerHand(CommunityCardSet cc, Deck gameDeck) {
38         for (int i = 0; i < HAND_LENGTH; i++){
39             communityCards.add(cc.getIthCard(i));
40             masterHand.add(cc.getIthCard(i));
```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/StudPokerHand.java

```

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
72 * master hand which contains seven card hand
73 * containing both community cards and two cards in stud hand
74 * @return array list of all 21 possible poker hands made from seven
75 * cards
76 */
77 private ArrayList<PokerHand> getAllFiveCardHands(){
78     for (int i = 0; i < STUD_HAND_LENGTH; i++ ){
79         for (int j = i+1; j < STUD_HAND_LENGTH; j++){
80             Card card1 = masterHand.get(i);

```

```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/StudPokerHand.java

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
93 getAllFiveCardHands method by using compareTo method
94 * in PokerHand class
95 * @return PokerHand that is the best possible poker hand out of all
96 21 poker hands
97 */
98 private PokerHand getBestFiveCardHand() {
99     ArrayList<PokerHand> hands = getAllFiveCardHands();
100    PokerHand bestSoFar = hands.get(0);
101    for (int i = 1; i < hands.size(); i++) {
102        if (hands.get(i).compareTo(bestSoFar) > 0) {
103            bestSoFar = hands.get(i);
104        }
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
111 comparison.
112 *
113 * @param other The hand to compare this hand to
114 * @return a negative number if this is worth LESS than other, zero
115 * if they are worth the SAME, and a positive number if this is worth
116 * MORE than other
117 */
118 public int compareTo(StudPokerHand other){

```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/StudPokerHand.java

```
118     PokerHand hand1 = getBestFiveCardHand();
119     PokerHand hand2 = other.getBestFiveCardHand();
120     return hand1.compareTo(hand2);
121 }
122 /**
123 * returns readable string version of stud cards
124 */
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 }
```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/PokerHandTests.java

```
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"));
```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/PokerHandTests.java

```

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

```

```
164     Testing.assertEquals("test of hand1 winning with high card", 1,
165     actual);
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,
184     actual);
185 }
186
187 public static void testTwoFlushesHand1Wins() {
188     ArrayList<Card> cardList1 = new ArrayList<Card>();
189     ArrayList<Card> cardList2 = new ArrayList<Card>();
190     cardList1.add(new Card("4", "C"));
191     cardList1.add(new Card("5", "C"));
192     cardList1.add(new Card("6", "C"));
193     cardList1.add(new Card("7", "C"));
194     cardList1.add(new Card("8", "C"));
195     PokerHand hand1 = new PokerHand(cardList1);
196     cardList2.add(new Card("2", "H"));
197     cardList2.add(new Card("4", "H"));
198     cardList2.add(new Card("4", "H"));
199     cardList2.add(new Card("5", "H"));
200     cardList2.add(new Card("6", "H"));
201     PokerHand hand2 = new PokerHand(cardList2);
202     int actual = hand1.compareTo(hand2);
203     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
222         flushes", -1, actual);
223     }
224
225     public static void testTwoTwoPairsHand1Wins() {
226         ArrayList<Card> cardList1 = new ArrayList<Card>();
227         ArrayList<Card> cardList2 = new ArrayList<Card>();
228         cardList1.add(new Card("6", "C"));
229         cardList1.add(new Card("6", "D"));
230         cardList1.add(new Card("13", "S"));
231         cardList1.add(new Card("13", "H"));
232         cardList1.add(new Card("2", "D"));
233         PokerHand hand1 = new PokerHand(cardList1);
234         cardList2.add(new Card("4", "C"));
235         cardList2.add(new Card("4", "H"));
236         cardList2.add(new Card("10", "D"));
237         cardList2.add(new Card("10", "C"));
238         cardList2.add(new Card("8", "C"));
239         PokerHand hand2 = new PokerHand(cardList2);
240         int actual = hand1.compareTo(hand2);
241         Testing.assertEquals("test of hand1 winning if both hands had
242         two pairs", 1, actual);
243     }
```

```
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"));
```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/PokerHandTests.java

```

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
335   card hands", -1, actual);
336 }
337 public static void testTies() {
338     ArrayList<Card> cardList1 = new ArrayList<Card>();
339     ArrayList<Card> cardList2 = new ArrayList<Card>();
340     cardList1.add(new Card("2", "H"));
341     cardList1.add(new Card("3", "C"));
342     cardList1.add(new Card("4", "S"));
343     cardList1.add(new Card("5", "C"));
344     cardList1.add(new Card("6", "C"));
345     PokerHand hand1 = new PokerHand(cardList1);
346     cardList2.add(new Card("2", "C"));
347     cardList2.add(new Card("3", "H"));
348     cardList2.add(new Card("4", "D"));
349     cardList2.add(new Card("5", "H"));
350     cardList2.add(new Card("6", "S"));
351     PokerHand hand2 = new PokerHand(cardList2);
352     int actual = hand1.compareTo(hand2);
353     Testing.assertEquals("test of tie between two hands", 0, actual);
354 }
355
356

```

```
1 package proj4;
2 import java.util.ArrayList;
3
4 public class CommunityCardSet {
5     final int HAND_LENGTH = 5;
6     private ArrayList<Card> communityHand = new ArrayList<Card>();
7
8     /**
9      * constructor for Community Card Set
10     * @param cardList takes list of Cards to add to Community Card Set
11     */
12    public CommunityCardSet(ArrayList<Card> cardList){
13        for (int i = 0; i < cardList.size(); i++) {
14            Card card = cardList.get(i);
15            communityHand.add(card);
16        }
17    }
18
19    /**
20     * constructor for Community Card Set
21     * @param gameDeck takes deck of cards and adds first five to
22     * Community Card Set
23     */
24    public CommunityCardSet(Deck gameDeck) {
25        for (int i = 0; i < HAND_LENGTH; i++){
26            Card card = gameDeck.deal();
27            communityHand.add(card);
28        }
29
30        /**
31         * checks if poker hand has standard amount of five cards, and adds
32         * new card only if poker hand has less than five cards
33         * @param card representation of playing card to add to hand
34         */
35        public void addCard(Card card) {
36            if (communityHand.size() < HAND_LENGTH) {
37                communityHand.add(card);
38            }
39
40        /**

```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/CommunityCardSet.java

```
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 * returns readable string version of community hand
53 */
54 public String toString(){
55     String toReturn = "";
56     for (int i = 0; i < HAND_LENGTH; i++){
57         Card card = communityHand.get(i);
58         toReturn = (toReturn + "|" + card.toString() + "|");
59     }
60     return toReturn;
61 }
62 }
63
64
65 }
66
```

```
1 package proj4;
2 import java.util.ArrayList;
3
4 public class CommunityCardTests {
5     private static final int COMMUNITY_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.getlthCard(COMMUNITY_HAND_LENGTH-1);
23        boolean actual = lastCardInCC.equals(toAdd);
24        Testing.assertEquals("Testing if add card function adds card to
25            end of community card hand when only starting with four cards",
26            true, actual);
27    }
28
29    public static void testingNoAddCard(){
30        ArrayList<Card> cardList = new ArrayList<Card>();
31        cardList.add(new Card("4", "C"));
32        cardList.add(new Card("5", "C"));
33        cardList.add(new Card("6", "H"));
34        cardList.add(new Card("7", "S"));
35        cardList.add(new Card("9", "C"));
36        CommunityCardSet cc = new CommunityCardSet(cardList);
37        Card toAdd = new Card("8", "D");
38        cc.addCard(toAdd);
39        Card lastCardInCC= cc.getlthCard(COMMUNITY_HAND_LENGTH-1);
        boolean actual = lastCardInCC.equals(toAdd);
        Testing.assertEquals("Testing if add card function stops adding
            new cards if Community Card Set already has five cards", false, actual)
```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/CommunityCardTests.java

```
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"));
```

```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/StudPokerHandTests.java
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 }
95 public static void testPairHand2Wins() {
96     ArrayList<Card> cardList = new ArrayList<Card>();
97     ArrayList<Card> hand1 = new ArrayList<Card>();
98     ArrayList<Card> hand2 = new ArrayList<Card>();
99     cardList.add(new Card("6", "H"));
100    cardList.add(new Card("9", "D"));
101    cardList.add(new Card("8", "C"));
102    cardList.add(new Card("3", "C"));
103    cardList.add(new Card("10", "S"));
104    CommunityCardSet cc = new CommunityCardSet(cardList);
105    hand1.add(new Card("14", "H"));
106    hand1.add(new Card("7", "D"));
107    hand2.add(new Card("12", "C"));
108    hand2.add(new Card("6", "S"));
109    StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
110    StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
111    int actual = studHand1.compareTo(studHand2);
112    Testing.assertEquals("test of hand2 winning with a pair", -1, actual
113 );
114 }
115 public static void testTwoPairHand1Wins() {
116     ArrayList<Card> cardList = new ArrayList<Card>();
117     ArrayList<Card> hand1 = new ArrayList<Card>();
118     ArrayList<Card> hand2 = new ArrayList<Card>();
119     cardList.add(new Card("9", "H"));
120     cardList.add(new Card("7", "D"));
121     cardList.add(new Card("8", "C"));
122     cardList.add(new Card("11", "C"));
123     cardList.add(new Card("3", "S"));
124     CommunityCardSet cc = new CommunityCardSet(cardList);
125     hand1.add(new Card("11", "H"));
126     hand1.add(new Card("7", "D"));
```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/StudPokerHandTests.java

```
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,
131     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,
150     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,
169     actual);
170 }
171 public static void testHighCardHand2Wins() {
172     ArrayList<Card> cardList = new ArrayList<Card>();
173     ArrayList<Card> hand1 = new ArrayList<Card>();
174     ArrayList<Card> hand2 = new ArrayList<Card>();
175     cardList.add(new Card("10", "H"));
176     cardList.add(new Card("9", "D"));
177     cardList.add(new Card("8", "C"));
178     cardList.add(new Card("11", "C"));
179     cardList.add(new Card("6", "S"));
180     CommunityCardSet cc = new CommunityCardSet(cardList);
181     hand1.add(new Card("4", "H"));
182     hand1.add(new Card("7", "D"));
183     hand2.add(new Card("12", "C"));
184     hand2.add(new Card("5", "S"));
185     StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
186     StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
187     int actual = studHand1.compareTo(studHand2);
188     Testing.assertEquals("test of hand2 winning with high card", -1,
189     actual);
190 }
191 public static void testTwoFlushesHand1Wins() {
192     ArrayList<Card> cardList = new ArrayList<Card>();
193     ArrayList<Card> hand1 = new ArrayList<Card>();
194     ArrayList<Card> hand2 = new ArrayList<Card>();
195     cardList.add(new Card("9", "H"));
196     cardList.add(new Card("7", "C"));
197     cardList.add(new Card("8", "C"));
198     cardList.add(new Card("11", "C"));
199     cardList.add(new Card("3", "S"));
200     CommunityCardSet cc = new CommunityCardSet(cardList);
201     hand1.add(new Card("14", "C"));
202     hand1.add(new Card("8", "C"));
203     hand2.add(new Card("12", "C"));
204     hand2.add(new Card("6", "C"));
205     StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/StudPokerHandTests.java

```

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

```

```

245     Testing.assertEquals("test of hand1 winning if both hands had
246     two pairs", 1, actual);
247 }
248 public static void testTwoTwoPairsHand2Wins() {
249     ArrayList<Card> cardList = new ArrayList<Card>();
250     ArrayList<Card> hand1 = new ArrayList<Card>();
251     ArrayList<Card> hand2 = new ArrayList<Card>();
252     cardList.add(new Card("9", "H"));
253     cardList.add(new Card("7", "D"));
254     cardList.add(new Card("8", "C"));
255     cardList.add(new Card("11", "C"));
256     cardList.add(new Card("3", "S"));
257     CommunityCardSet cc = new CommunityCardSet(cardList);
258     hand1.add(new Card("7", "H"));
259     hand1.add(new Card("9", "D"));
260     hand2.add(new Card("11", "S"));
261     hand2.add(new Card("9", "S"));
262     StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
263     StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
264     int actual = studHand1.compareTo(studHand2);
265     Testing.assertEquals("test of hand2 winning if both hands had
266     two pairs", -1, actual);
267 }
268 public static void testTwoPairsHand1Wins() {
269     ArrayList<Card> cardList = new ArrayList<Card>();
270     ArrayList<Card> hand1 = new ArrayList<Card>();
271     ArrayList<Card> hand2 = new ArrayList<Card>();
272     cardList.add(new Card("9", "H"));
273     cardList.add(new Card("7", "D"));
274     cardList.add(new Card("8", "C"));
275     cardList.add(new Card("11", "C"));
276     cardList.add(new Card("3", "S"));
277     CommunityCardSet cc = new CommunityCardSet(cardList);
278     hand1.add(new Card("14", "H"));
279     hand1.add(new Card("7", "S"));
280     hand2.add(new Card("12", "C"));
281     hand2.add(new Card("7", "C"));
282     StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
283     StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
284     int actual = studHand1.compareTo(studHand2);
285     Testing.assertEquals("test of hand1 winning if both hands had a
286     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
303                            pair", -1, actual);
304    }
305    public static void testTwoHighCardsHand1Wins() {
306        ArrayList<Card> cardList = new ArrayList<Card>();
307        ArrayList<Card> hand1 = new ArrayList<Card>();
308        ArrayList<Card> hand2 = new ArrayList<Card>();
309        cardList.add(new Card("9", "H"));
310        cardList.add(new Card("7", "D"));
311        cardList.add(new Card("8", "C"));
312        cardList.add(new Card("11", "C"));
313        cardList.add(new Card("3", "S"));
314        CommunityCardSet cc = new CommunityCardSet(cardList);
315        hand1.add(new Card("14", "H"));
316        hand1.add(new Card("5", "D"));
317        hand2.add(new Card("12", "C"));
318        hand2.add(new Card("6", "S"));
319        StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
320        StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
321        int actual = studHand1.compareTo(studHand2);
322        Testing.assertEquals("test of hand1 winning if both were high
323                            card hands", 1, actual);
324    }
325    public static void testTwoHighCardsHand2Wins() {
```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/StudPokerHandTests.java

```
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
341   card hands", -1, actual);
342 }
343 public static void testTies() {
344     ArrayList<Card> cardList = new ArrayList<Card>();
345     ArrayList<Card> hand1 = new ArrayList<Card>();
346     ArrayList<Card> hand2 = new ArrayList<Card>();
347     cardList.add(new Card("9", "H"));
348     cardList.add(new Card("7", "D"));
349     cardList.add(new Card("8", "C"));
350     cardList.add(new Card("11", "C"));
351     cardList.add(new Card("10", "S"));
352     CommunityCardSet cc = new CommunityCardSet(cardList);
353     hand1.add(new Card("4", "H"));
354     hand1.add(new Card("5", "D"));
355     hand2.add(new Card("2", "C"));
356     hand2.add(new Card("6", "S"));
357     StudPokerHand studHand1 = new StudPokerHand(cc, hand1);
358     StudPokerHand studHand2 = new StudPokerHand(cc, hand2);
359     int actual = studHand1.compareTo(studHand2);
360     Testing.assertEquals("test of tie using all community cards
361   between two hands", 0, actual);
362 }
363 public static void testTies2() {
364     ArrayList<Card> cardList = new ArrayList<Card>();
365     ArrayList<Card> hand1 = new ArrayList<Card>();
```

File - /Users/lilahkelly/Downloads/LilahKellyProject4/src/proj4/StudPokerHandTests.java

```
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
```