

```

1  package proj4; // do not erase. Gradescope expects this.
2
3  import java.util.HashMap;
4  import java.util.Map;
5
6  public class Card {
7
8
9      private int cardRank; // Rank of the card
10     private String cardSuit; // Suit of the card
11
12     /**
13      * Card Object Constructor
14      * @param rank rank of the card
15      * @param suit suit of the card
16      */
17     public Card(int rank, String suit){
18
19         cardRank = rank;
20         cardSuit = suit;
21
22     }
23
24     /** * constructor
25      * * @param rank integer between 2-14
26      * * @param suit integer: 0=Spades, 1=Hearts, 2=Clubs
27      * * or 3=Diamonds
28      */
29     public Card(int rank, int suit){
30
31         Map<Integer, String> suitMap = new HashMap<>();
32         suitMap.put(0, "Spades");
33         suitMap.put(1, "Hearts");
34         suitMap.put(2, "Clubs");
35         suitMap.put(3, "Diamonds");
36
37         if(suitMap.containsKey(suit)){
38             cardSuit = suitMap.get(suit);
39         }
40         else{
41             throw new IllegalArgumentException("suit must
42             be represented with an int 0-3");
43         }
44
45         cardRank =rank;
46     }
47
48     /** * constructor
49      * * @param rank String: whole cards (2-10) can either
50      * * be spelled
51      * * out like "two" or numeric like "2". Case

```

```

49 insensitive.
50     * * @param suit String: "Spades", "Hearts", "Clubs
    ", or "Diamonds"
51     * */
52     public Card(String rank, String suit){
53
54         Map<String, Integer> rankMap = new HashMap<>();
55         rankMap.put("two", 2);
56         rankMap.put("three", 3);
57         rankMap.put("four", 4);
58         rankMap.put("five", 5);
59         rankMap.put("six", 6);
60         rankMap.put("seven", 7);
61         rankMap.put("eight", 8);
62         rankMap.put("nine", 9);
63         rankMap.put("ten", 10);
64         rankMap.put("eleven", 11);
65         rankMap.put("twelve", 12);
66         rankMap.put("thirteen", 13);
67         rankMap.put("fourteen", 14);
68         rankMap.put("Two", 2);
69         rankMap.put("Three", 3);
70         rankMap.put("Four", 4);
71         rankMap.put("Five", 5);
72         rankMap.put("Six", 6);
73         rankMap.put("Seven", 7);
74         rankMap.put("Eight", 8);
75         rankMap.put("Nine", 9);
76         rankMap.put("Ten", 10);
77         rankMap.put("Eleven", 11);
78         rankMap.put("Twelve", 12);
79         rankMap.put("Thirteen", 13);
80         rankMap.put("Fourteen", 14);
81         rankMap.put("jack", 11);
82         rankMap.put("queen", 12);
83         rankMap.put("king", 13);
84         rankMap.put("ace", 14);
85         rankMap.put("Jack", 11);
86         rankMap.put("Queen", 12);
87         rankMap.put("King", 13);
88         rankMap.put("Ace", 14);
89
90
91         if(rankMap.containsKey(rank)){
92             cardRank = rankMap.get(rank);
93         }
94         else{
95             cardRank = Integer.parseInt(rank);
96         }
97
98         cardSuit = suit;
99     }

```

```
100
101
102     /**
103      * getter for the rank of the card
104      * @return int: rank
105      */
106     public int getCardRank() {
107         return cardRank;
108     }
109
110     /**
111      * Getter for the suit of the card
112      *
113      * @return String: suit
114      */
115     public String getCardSuit() {
116         return cardSuit;
117     }
118
119     /**
120      * format the Card as a string for printing and such
121      *
122      * @return String of the card
123      */
124     public @Override String toString() {
125
126         String rankHolder = Integer.toString(cardRank);
127         if(cardRank == 11) {
128             rankHolder = "Jack";
129         }
130         if(cardRank == 12) {
131             rankHolder = "Queen";
132         }
133         if(cardRank == 13) {
134             rankHolder = "King";
135         }
136         if(cardRank == 14) {
137             rankHolder = "Ace";
138         }
139
140         String cardString = rankHolder + " of " +
cardSuit;
141         return cardString;
142     }
143 }
144
145 }
146
```

```

1  package proj4; // do not erase. Gradescope expects this.
2
3
4  import java.util.*;
5  import java.util.concurrent.ThreadLocalRandom;
6
7  /**
8   * Class for modeling a Deck of Cards
9   */
10 public class Deck {
11
12     private final int DECK_SIZE = 52; // standard deck size
13     private ArrayList<Card> cardList; // ArrayList of cards
14     private int nextToDeal; // index we are drawing next
15     card from
16
17     /**
18      * Constructor for a Deck object
19      */
20     public Deck() {
21         nextToDeal = 0;
22         cardList = new ArrayList<Card>(DECK_SIZE);
23
24         generateDeck(); // generates the deck
25     }
26
27     /**
28      * Generates a standard 52 card deck
29      * ranks: 2 to 14
30      * suits: Hearts, Diamonds, Spades, Clubs
31      */
32     private void generateDeck() {
33
34
35         Map<Integer, String> suitMap = new HashMap<>();
36         suitMap.put(0, "Spades");
37         suitMap.put(1, "Hearts");
38         suitMap.put(2, "Clubs");
39         suitMap.put(3, "Diamonds");
40
41         int[] ranks = new int[] {2, 3, 4, 5, 6, 7, 8, 9,
42     10, 11, 12, 13, 14};
43
44         for(Integer key: suitMap.keySet()){
45             for(int j : ranks) {
46                 Card card = new Card(j, key);
47                 cardList.add(card);
48             }
49         }
50

```

```

51
52     /**
53      * shuffles the deck of cards
54      */
55     public void shuffle() {
56
57         for(int i = nextToDeal; i < cardList.size(); i
58 ++){
59             Card currentCard = cardList.get(i);
60
61             int random = ThreadLocalRandom.current().
62 nextInt(i, cardList.size());
63
64             Card swapCard = cardList.get(random);
65
66             cardList.set(i, swapCard);
67             cardList.set(random, currentCard);
68         }
69
70
71     /**
72      * deals the next card in the deck
73      * efficiency: O(1)
74      * @return previously undelt Card
75      */
76     public Card deal() {
77         if(nextToDeal == this.size()){
78             return null;
79         }
80         else {
81             Card topCard = cardList.get(nextToDeal);
82             nextToDeal++;
83             return topCard;
84         }
85
86     }
87
88     /**
89      * return size of the deck; number of undelt cards
90      * @return int value of number of cards
91      */
92     public int size() {
93         int deckSize;
94         deckSize = cardList.size() - nextToDeal;
95         return deckSize;
96     }
97
98     /**
99      * reset the next card to deal to the first in the
100 deck

```

```
100     */
101     public void gather() {
102         nextToDeal = 0;
103     }
104
105     /**
106      * format the deck as a string for printing and such
107      * @return String
108      */
109     public @Override String toString() {
110         String str = "";
111         for (int i = nextToDeal; i < cardList.size(); i++) {
112             str += cardList.get(i).toString();
113             str += "\n";
114         }
115         return str;
116     }
117
118     /**
119      * determines if there are cards left in the deck or
120      * not
121      * @return boolean. True if no more cards, false
122      * otherwise
123      */
124     public boolean isEmpty() {
125         if (nextToDeal == DECK_SIZE) {
126             return true;
127         }
128         else {
129             return false;
130         }
131     }
132 }
```

```

1  package proj4;
2
3  import java.util.ArrayList;
4  import java.util.Scanner;
5
6  /**
7   * Author: Ian Sulley
8   *
9   * Honor Code: I affirm that I have carried out the
10  attached academic endeavors with full academic honesty,
11  * in accordance with the Union College Honor Code and the
12  course syllabus
13  */
14
15  public class Client{
16
17      public static void main(String[] args){
18
19          boolean isOver = false;
20
21          Deck myDeck = new Deck();
22          myDeck.shuffle();
23
24          int playerScore = 0;
25
26          while(myDeck.size() > 9 && !isOver){
27
28              ArrayList<Card> cardArrayListCC = new
29  ArrayList<Card>();
30              for(int i = 0; i < 5; i++){
31                  cardArrayListCC.add(myDeck.deal());
32              }
33
34              ArrayList<Card> Hand1Cards = new ArrayList
35  <>();
36              ArrayList<Card> Hand2Cards = new ArrayList
37  <>();
38              for(int i = 0; i < 2; i++){
39                  Hand1Cards.add(myDeck.deal());
40                  Hand2Cards.add(myDeck.deal());
41              }
42
43              CommunityCardSet communityCards = new
44  CommunityCardSet(cardArrayListCC);
45              StudPokerHand hand1 = new StudPokerHand(
46  communityCards, Hand1Cards);
47              StudPokerHand hand2 = new StudPokerHand(
48  communityCards, Hand2Cards);
49
50              System.out.println(hand1);

```

```
45         System.out.println(hand2);
46
47         Scanner input = new Scanner(System.in);
48         System.out.println("Which hand wins (enter 1
for the first hand, 2 for the second hand, or 0 for tie);
49         int userGuess = input.nextInt();
50
51
52         while(userGuess != 1 && userGuess != 2 &&
userGuess != 0){
53             System.out.println("Invalid entry, please
try 1, 2 or a space:");
54             userGuess = input.nextInt();
55         }
56
57
58         System.out.print("Your input:");
59         System.out.print(input);
60
61         int expectedAnswer = hand1.compareTo(hand2);
62
63         if(expectedAnswer == userGuess || (
expectedAnswer == 0 && userGuess == 0)){
64             playerScore++;
65             System.out.println("Congrats! You are
correct. +1 point");
66         }
67         else{
68             System.out.println("Sorry, wrong answer");
69             isOver = false;
70         }
71     }
72     System.out.println("Game over, your score is: ");
73     System.out.print(playerScore);
74 }
75 }
```



```

1  package proj4;
2
3  /**
4   * This class contains a collection of methods that help
   * with testing. All methods
5   * here are static so there's no need to construct a
   * Testing object. Just call them
6   * with the class name like so:
7   * 

<p></p>


8   * Testing.assertEquals("test description", expected
   * , actual)</code>
9   *
10  * @author Kristina Striegnitz, Aaron Cass, Chris
   * Fernandes
11  * @version 5/28/18
12  */
13  public class Testing {
14
15      private static boolean VERBOSE = false;
16      private static int numTests;
17      private static int numFails;
18
19      /**
20       * Toggles between a lot of output and little output.
21       *
22       * @param verbose
23       * If verbose is true, then complete
   * information is printed,
24       * whether the tests passes or fails. If
   * verbose is false, only
25       * failures are printed.
26       */
27      public static void setVerbose(boolean verbose)
28      {
29          VERBOSE = verbose;
30      }
31
32      /**
33       * Each of the assertEquals methods tests whether the
   * actual
34       * result equals the expected result. If it does, then
   * the test
35       * passes, otherwise it fails.
36       *
37       * The only difference between these methods is the
   * types of the
38       * parameters.
39       *
40       * All take a String message and two values of some
   * other type to
41       * compare:
42       */

```

```

43      * @param message
44      *           a message or description of the test
45      * @param expected
46      *           the correct, or expected, value
47      * @param actual
48      *           the actual value
49      */
50      public static void assertEquals(String message,
boolean expected,
51                                     boolean actual)
52      {
53          printTestCaseInfo(message, "\"" + expected, "\"" +
actual);
54          if (expected == actual) {
55              pass();
56          } else {
57              fail(message);
58          }
59      }
60
61      public static void assertEquals(String message, int
expected, int actual)
62      {
63          printTestCaseInfo(message, "\"" + expected, "\"" +
actual);
64          if (expected == actual) {
65              pass();
66          } else {
67              fail(message);
68          }
69      }
70
71      public static void assertEquals(String message, Object
expected,
72                                     Object actual)
73      {
74          String expectedString = "<<null>>";
75          String actualString = "<<null>>";
76          if (expected != null) {
77              expectedString = expected.toString();
78          }
79          if (actual != null) {
80              actualString = actual.toString();
81          }
82          printTestCaseInfo(message, expectedString,
actualString);
83
84          if (expected == null) {
85              if (actual == null) {
86                  pass();
87              } else {
88                  fail(message);

```

```

89         }
90     } else if (expected.equals(actual)) {
91         pass();
92     } else {
93         fail(message);
94     }
95 }
96
97 /**
98  * Asserts that a given boolean must be true. The
test fails if
99  * the boolean is not true.
100  *
101  * @param message The test message
102  * @param actual The boolean value asserted to be
true.
103  */
104 public static void assertTrue(String message, boolean
actual)
105 {
106     assertEquals(message, true, actual);
107 }
108
109 /**
110  * Asserts that a given boolean must be false. The
test fails if
111  * the boolean is not false (i.e. if it is true).
112  *
113  * @param message The test message
114  * @param actual The boolean value asserted to be
false.
115  */
116 public static void assertFalse(String message,
boolean actual)
117 {
118     assertEquals(message, false, actual);
119 }
120
121 private static void printTestCaseInfo(String message
, String expected,
122                                     String actual)
123 {
124     if (VERBOSE) {
125         System.out.println(message + ":");
126         System.out.println("expected: " + expected);
127         System.out.println("actual:   " + actual);
128     }
129 }
130
131 private static void pass()
132 {
133     numTests++;

```

```

134
135         if (VERBOSE) {
136             System.out.println("--PASS--");
137             System.out.println();
138         }
139     }
140
141     private static void fail(String description)
142     {
143         numTests++;
144         numFails++;
145
146         if (!VERBOSE) {
147             System.out.print(description + " ");
148         }
149         System.out.println("--FAIL--");
150         System.out.println();
151     }
152
153     /**
154      * Prints a header for a section of tests.
155      *
156      * @param sectionTitle The header that should be
157      * printed.
158      */
159     public static void testSection(String sectionTitle)
160     {
161         if (VERBOSE) {
162             int dashCount = sectionTitle.length();
163             System.out.println(sectionTitle);
164             for (int i = 0; i < dashCount; i++) {
165                 System.out.print("-");
166             }
167             System.out.println();
168             System.out.println();
169         }
170     }
171
172     /**
173      * Initializes the test suite. Should be called
174      * before running any
175      * tests, so that passes and fails are correctly
176      * tallied.
177      */
178     public static void startTests()
179     {
180         System.out.println("Starting Tests");
181         System.out.println();
182         numTests = 0;
183         numFails = 0;
184     }

```

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

```
1 package proj4;
2
3
4 public class CardTests{
5
6     public static void testAll(){
7         testGetCardRank();
8         testGetCardSuit();
9         testGetCardSuit2();
10        testCardToString();
11    }
12
13    public static void testGetCardRank(){
14
15        Card myCard = new Card(4, 2);
16
17        int expectedValue = 4;
18        int actualValue = myCard.getCardRank();
19
20        Testing.assertEquals("Testing getCardRank",
21            expectedValue,
22            actualValue);
23    }
24
25    public static void testGetCardSuit(){
26
27        Card myCard = new Card(4, 2);
28
29        String expectedValue = "Clubs";
30        String actualValue = myCard.getCardSuit();
31
32        Testing.assertEquals("Testing getCardSuit",
33            expectedValue,
34            actualValue);
35    }
36    public static void testGetCardSuit2(){
37
38        Card myCard = new Card(4, "Spades");
39
40        String expectedValue = "Spades";
41        String actualValue = myCard.getCardSuit();
42
43        Testing.assertEquals("Testing getCardSuit",
44            expectedValue,
45            actualValue);
46    }
47
48    public static void testCardToString(){
49
50        Card myCard = new Card(4, 2);
51
52        String expectedValue = "[4 of Clubs]";
```

```
53         String actualValue = myCard.toString();
54
55         Testing.assertEquals("Testing getCardSuit",
56                             expectedValue,
57                             actualValue);
58     }
59 }
60
61
```

```
1 package proj4;  
2  
3 public class DeckTests{  
4  
5 }
```



```

1  package proj4; // do not erase. Gradescope expects this.
2
3  import java.util.*;
4
5  public class PokerHand {
6
7      private static final int MAX_HAND_SIZE = 5;
8      private ArrayList<Card> cardsInHand; //all the cards
        in the hand
9
10     /**
11      * A Constructor for a PokerHand Object
12      *
13      * @param cardList cards that will make up the
        PokerHand
14      */
15     public PokerHand(ArrayList<Card> cardList) {
16         cardsInHand = cardList;
17     }
18
19     /**
20      * add a card to the Poker Hand if there are less than
        5 cards in the hand
21      * otherwise do nothing
22      *
23      * @param card card being added to the PokerHand
24      */
25     public void addCard(Card card) {
26
27         if (cardsInHand.size() < MAX_HAND_SIZE) {
28             cardsInHand.add(card);
29         }
30     }
31
32     /**
33      * return the card in the pokerHand at the given index
34      *
35      * @param index index of card being retrieved
36      * @return Card
37      */
38     public Card get_ith_card(int index) {
39         if (index >= 0 && index < cardsInHand.size()) {
40             return cardsInHand.get(index);
41         } else {
42             return null;
43         }
44     }
45
46     /**
47      * override the toString function to turn a PokerHand
        into a properly formatted string
48      * @return String

```

```

49      */
50      public @Override String toString() {
51          String str = "";
52          for (int i = 0; i < cardsInHand.size(); i++) {
53              str += cardsInHand.get(i).toString();
54              str += "\n";
55          }
56          return str;
57      }
58
59      /**
60       * Determines how this hand compares to another hand
61       * , returns
62       * * positive, negative, or zero depending on the
63       * comparison.
64       * @param other The hand to compare this hand to
65       * @return a negative number if this is worth LESS
66       * than other, zero
67       * * if they are worth the SAME, and a positive number
68       * if this is worth
69       * * MORE than other
70       */
71      public int compareTo(PokerHand other) {
72          //organize the hands and determine their types//
73          //THIS hand
74          TreeMap<Integer, Integer> rankOccurrences = this.
75          sortRanks();
76          ArrayList<Integer> pairRanks = getRanks(
77          rankOccurrences, 1); //for seperating out the pairs
78          ArrayList<Integer> highcardRanks = getRanks(
79          rankOccurrences, 0) ; //for seperating out the non-pairs
80          Integer hand1Type = this.handType(pairRanks.size
81          ());
82          //OTHER hand
83          TreeMap<Integer, Integer> otherRankOccurrences =
84          other.sortRanks();
85          ArrayList<Integer> otherPairRanks = getRanks(
86          otherRankOccurrences, 1); //for seperating out the pairs
87          ArrayList<Integer> otherHighcardRanks = getRanks(
88          otherRankOccurrences, 0) ; //for seperating out the non-
89          pairs
90          Integer hand2Type = other.handType(otherPairRanks
91          .size());
92
93          int handTypeComparison = hand1Type.compareTo(
94          hand2Type);
95
96          if(handTypeComparison != 0){

```

```

87         return handTypeComparison;
88     }
89
90     else { //if hands are of the same type...
91         if(!pairRanks.isEmpty()){ //if there are
pairs to compare...
92             int pairCompare = this.tieBreaker(
pairRanks, otherPairRanks); //compare them
93             if(pairCompare == 0){ //if the pair
values are equal
94                 return this.tieBreaker(highcardRanks
, otherHighcardRanks); // return the highcard comparison
95             }
96             else{ //otherwise return the pair
comparison
97                 return pairCompare;
98             }
99         }
100     }
101     else{ //if there are no pairs to compare,
just return the highcard comparison
102         return this.tieBreaker(highcardRanks,
otherHighcardRanks);
103     }
104 }
105 }
106
107 /**
108  * Determing the type of the hand. Flush, 2pair,
1pair, or highcard
109  *
110  * @return Integer : 4 if flush, 3 if 2pair, 2 if
1pair, 1 if highcard
111  */
112 private Integer handType(int pairRanksSize) {
113
114     boolean isFlush = flushCheck();
115     if(isFlush) {
116         return 4; //FLUSH
117     }
118     if (pairRanksSize == 2) { // if you have 2 pairs
119         return 3; //2Pair
120     }
121     if (pairRanksSize == 1) { //if you have 1 pair
122         return 2; //1 PAIR
123     }
124     else { //If its not a flush, 2pair, or 1pair it
has to be....
125         return 1; //HIGHCARD
126     }
127 }
128 }

```

```

129
130     /**
131      * creates a Treemap of the ranks and their # of
132      * occurrences
133      * @return TreeMap<Integer, Integer>
134      */
135     private TreeMap<Integer, Integer> sortRanks(){
136
137         ArrayList<Integer> allRanks = new ArrayList<
138 Integer>(); //for seperating out the ranks of the cards
139         TreeMap<Integer, Integer> rankOccurrences = new
140 TreeMap<Integer, Integer>(Collections.reverseOrder()); //
141 experimenting with a new data structure, makes sorting
142 fuctions simplier
143
144         for (int i = 0; i < cardsInHand.size(); i++) {
145             Card currentCard = cardsInHand.get(i); //
146             current card we are pulling data from
147             allRanks.add(currentCard.getCardRank()); //
148             adding the current cards rank to the rank array
149         }
150
151         //sort allRanks into a TreeMap with Key = Rank &
152 Value = instances of the rank
153         for (Integer i : allRanks) {
154             Integer j = rankOccurrences.get(i);
155             rankOccurrences.put(i, (j == null) ? 1 : j + 1
156 );
157         }
158         return rankOccurrences;
159     }
160
161     /**
162      * sorts all the ranks into pairs and non pairs from
163      * the treemap
164      * @param rankOccurrences treemap of ranks present and
165      * the # of occurrences of each
166      * @param whichRanks if we are sorting pairs (1) or
167      * non-pairs (0)
168      * @return
169      */
170     public ArrayList<Integer> getRanks(TreeMap<Integer,
171 Integer> rankOccurrences, int whichRanks){
172
173         ArrayList<Integer> pairRanks = new ArrayList<
174 Integer>(); //for seperating out the pairs
175         ArrayList<Integer> highcardRanks = new ArrayList<
176 Integer>(); //for seperating out the non-pairs
177
178         //sort rankOccurrences by pairs and non-pairs(
179 highcards)

```

```

165         for(Integer key : rankOccurrences.keySet()){
166             if(rankOccurrences.get(key) == 4){ //2pair
167                 pairRanks.add(key);
168                 pairRanks.add(key);
169             }
170             if(rankOccurrences.get(key) == 2 ||
rankOccurrences.get(key) == 3){
171                 pairRanks.add(key);
172             }
173             else{
174                 highcardRanks.add(key);
175             }
176         }
177
178         if(whichRanks == 0){
179             return highcardRanks;
180         }
181         if(whichRanks == 1){
182             return pairRanks;
183         }
184         else{
185             return null;
186         }
187     }
188     /**
189      * compares two ArrayLists of ranks and determines
which has the first instance of a greater value
190      *
191      * @param theseRanks ArrayList of ranks from this
hand
192      * @param otherRanks ArrayList of ranks from other
hand
193      * @return int 1 if theseRanks is greater, -1 if
otherRanks is greater, 0 if all ranks are the same
194      */
195     private int tieBreaker(ArrayList<Integer> theseRanks
, ArrayList<Integer> otherRanks) {
196
197         //compare each rank
198         for (int i = 0; i < theseRanks.size() && i <
otherRanks.size(); i++) {
199             int currentCompare = theseRanks.get(i).
compareTo(otherRanks.get(i)); //compare current index
200             if (currentCompare != 0) { //if the current
index ranks are different...
201                 return currentCompare; //return the
comparison
202             }
203         }
204         return 0; //you make it through all ranks and
they are all the same
205     }

```

```
206
207     /**
208     * checks if the hand is a flush
209     *
210     * @return true if hand is a flush, false if not
211     */
212     private boolean flushCheck(){
213
214         ArrayList<String> allSuits = new ArrayList<String>
215         >(); //for seperating out the suits
216
217         for (int i = 0; i < cardsInHand.size(); i++) {
218             Card currentCard = cardsInHand.get(i); //
219             current card we are pulling data from
220             allSuits.add(currentCard.getCardSuit()); //
221             adding the current cards suit to the suit array
222         }
223
224         //checking for a flush
225         String checkSuit = allSuits.get(1); //a suit
226         present in the hand
227         if(Collections.frequency(allSuits, checkSuit) ==
228         cardsInHand.size()){
229             return true;
230         }
231         else{
232             return false;
233         }
234     }
235 }
```

```

1  package proj4;
2  import java.util.*;
3
4
5  public class StudPokerHand{
6
7      private static final int MAX_HAND_SIZE = 2;
8      private ArrayList<Card> cardsInHand;
9      private CommunityCardSet communityCards;
10
11
12     /**
13      * Constructor for a StudPokerHand
14      * @param cc communityCard set for this hand
15      * @param cardArrayList the cards in this hand
16      */
17     public StudPokerHand(CommunityCardSet cc, ArrayList<
18 Card> cardArrayList){
19         cardsInHand = cardArrayList;
20         communityCards = cc;
21     }
22
23     /**
24      * return the card in the StudPokerHand at the given
25 index
26      * @param index index of card being retrieved
27      * @return Card
28      */
29     public Card get_ith_card(int index) {
30         if (index >= 0 && index < cardsInHand.size()) {
31             return cardsInHand.get(index);
32         } else {
33             return null;
34         }
35     }
36
37     /**
38      * add a card to the StudPokerHand if there are less
39 than 2 cards in the hand
40      * otherwise do nothing
41      * @param card card being added to the PokerHand
42      */
43     public void addCard(Card card) {
44
45         if (cardsInHand.size() < MAX_HAND_SIZE) {
46             cardsInHand.add(card);
47         }
48     }
49

```

```

50
51     /**
52     * Determines how this hand compares to another hand
53     , using the
54     * community card set to determine the best 5-card
55     hand it can * make. Returns positive, negative, or zero
56     depending on the comparison.
57     * @param other The hand to compare this hand to
58     * * @return a negative number if this is worth LESS
59     than other, zero
60     * * if they are worth the SAME, and a positive
61     number if this is worth * MORE than other
62     * */
63     public int compareTo(StudPokerHand other){
64
65         PokerHand thisBestHand = this.getBestFiveCardHand
66         ();
67         PokerHand otherBestHand = other.
68         getBestFiveCardHand();
69
70         return thisBestHand.compareTo(otherBestHand);
71     }
72
73     /**
74     * override the toString function to turn a PokerHand
75     into a properly formatted string
76     * @return String
77     */
78     public @Override String toString(){
79         String studString = "The Community Cards are: ";
80         studString += communityCards.toString();
81         studString += "\n The Hole Cards are: ";
82
83         for (Card myCard:cardsInHand) {
84             studString += myCard.toString();
85             studString += " ";
86         }
87
88         return studString;
89     }
90
91     /**
92     * determines the best possible 5 card hand from all
93     possible 5 card hands
94     * @return PokerHand of highest evaluation
95     */
96     private PokerHand getBestFiveCardHand() {
97         ArrayList<PokerHand> hands = getAllFiveCardHands
98         ();
99         PokerHand bestSoFar = hands.get(0);
100         for (int i = 1; i < hands.size(); i++) {

```



```

92         if (hands.get(i).compareTo(bestSoFar) > 0) {
93             bestSoFar = hands.get(i);
94         }
95     }
96     return bestSoFar;
97 }
98
99 /**
100  * generates all possible five card hands from the
101  * community cards and the hole cards
102  * @return ArrayList of PokerHands
103  */
104     private ArrayList<PokerHand> getAllFiveCardHands(){
105         ArrayList<PokerHand> allHands = new ArrayList<
106         PokerHand>(); //keep track of all the hands
107         ArrayList<Card> allCards = new ArrayList<Card>(
108         cardsInHand); //keep track of all the cards in this hand
109         (hole cards + community cards)
110
111         for(int i = 0; i < communityCards.size(); i++){
112             allCards.add(communityCards.get_i_th_card(i));
113             //add all the community cards to all cards
114         }
115
116         for(int i = 0; i < communityCards.size() +
117         cardsInHand.size(); i++) { //these for loops just iterate
118             through every index in the list removing a different
119             combo of two cards
120
121             for(int j = i + 1; j < communityCards.size
122             () + cardsInHand.size() - 1; j++) { //which generates
123             all the unique 5 card hands
124
125                 ArrayList<Card> cloneAllCards = new
126                 ArrayList<Card>(allCards); // make a copy of allCards to
127                 remove from
128
129                 cloneAllCards.remove(i); //remove 1 card
130                 cloneAllCards.remove(j); //remove another
131
132                 PokerHand currentHand = new PokerHand(
133                 cloneAllCards); //make a hand with the remaining cards
134                 allHands.add(currentHand); //add the new
135                 hand to the list of hands
136             }
137         }
138         return allHands;
139     }
140 }
141
142
143
144
145
146

```

```

1  package proj4;
2
3  import java.util.*;
4
5  public class CommunityCardSet{
6
7      private ArrayList<Card> communityCards = new ArrayList
        <Card>(5);
8      private final int MAX_CC_SIZE = 5;
9      public CommunityCardSet(ArrayList<Card> cardList){
10         communityCards.addAll(cardList);
11     }
12
13
14     /**
15      * return the card in the pokerHand at the given index
16      *
17      * @param index index of card being retrieved
18      * @return Card
19      */
20     public Card get_ith_card(int index) {
21         if (index >= 0 && index < communityCards.size()) {
22             return communityCards.get(index);
23         } else {
24             return null;
25         }
26     }
27
28     public int size(){
29         return communityCards.size();
30     }
31
32     /**
33      * add a card to the Poker Hand if there are less than
34      5 cards in the hand
35      * otherwise do nothing
36      *
37      * @param card card being added to the PokerHand
38      */
39     public void addCard(Card card) {
40
41         if (communityCards.size() < MAX_CC_SIZE) {
42             communityCards.add(card);
43         }
44
45     }
46
47     public @Override String toString(){
48         String str = "";
49         for (int i = 0; i < communityCards.size(); i++) {
50             str += communityCards.get(i).toString();
51             str += "\n";
52         }
53     }

```

```
51         return str;
52     }
53 }
```

```

1  package proj4;
2
3  import java.util.ArrayList;
4  import java.util.Arrays;
5
6  public class StudPokerHandTests{
7
8      public static void main(String[] args) {
9
10         test_all();
11     }
12
13     public static void test_all(){
14
15         testSPHCompareTo();
16         testAddCard();
17     }
18
19
20     public static void testSPHCompareTo(){
21
22         CommunityCardSet cc = new CommunityCardSet(new
23         ArrayList<Card>(Arrays.asList(new Card(4, "C"), new Card(2
24         , "C"), new Card(7, "C"), new Card(5, "C"), new Card(10, "
25         C"))));
26
27         StudPokerHand hand1 = new StudPokerHand(cc, (new
28         ArrayList<Card> (Arrays.asList(new Card(13, "S"), new Card
29         (12, "S"))));
30
31         StudPokerHand hand2 = new StudPokerHand(cc, (new
32         ArrayList<Card> (Arrays.asList(new Card(10, "S"), new Card
33         (5, "H"))));
34
35         int expectedValue = -1;
36
37         int actualValue = hand1.compareTo(hand2);
38
39         Testing.assertEquals("Testing StudPokerHand
40         CompareTo",
41             expectedValue,
42             actualValue);
43     }
44
45     public static void testAddCard(){
46
47         CommunityCardSet cc = new CommunityCardSet(new
48         ArrayList<Card>(Arrays.asList(new Card(4, "C"), new Card(2
49         , "C"), new Card(7, "C"), new Card(5, "C"), new Card(10, "
50         C"))));
51
52         StudPokerHand hand1 = new StudPokerHand(cc, (new
53         ArrayList<Card> (Arrays.asList(new Card(10, "S"))));

```

```
41         hand1.addCard(new Card(5, "H"));
42
43         int expectedValue = 1;
44
45         int actualValue = 1;
46
47         Testing.assertEquals("Testing StudPokerHand
testAddCard",
48                             expectedValue,
49                             actualValue);
50
51
52
53     }
54
55     public static void testGetIthCard(){
56
57         CommunityCardSet cc = new CommunityCardSet(new
ArrayList<Card>(Arrays.asList(new Card(4, "C"), new Card(2
, "C"), new Card(7, "C"), new Card(5, "C"), new Card(10, "
C"))));
58         StudPokerHand hand1 = new StudPokerHand(cc, (new
ArrayList<Card> (Arrays.asList(new Card(10, "S"))));
59
60
61
62         int expectedValue = 1;
63
64         int actualValue = 1;
65
66         Testing.assertEquals("Testing StudPokerHand
testAddCard",
67                             expectedValue,
68                             actualValue);
69
70
71
72     }
73 }
```

```

1  package proj4;
2
3  import java.util.ArrayList;
4  import java.util.Arrays;
5
6  /**
7   * Author: Ian Sulley
8   *
9   * Honor Code: I affirm that I have carried out the
   attached academic endeavors
10  * with full academic honesty, in accordance with the
   Union College Honor Code
11  * and the course syllabus
12  */
13
14 /**
15  * Testing Class for PokerHand compareTo()
16  */
17 public class PokerComparisonTests {
18
19     public static void main(String[] args) {
20
21         test_all();
22     }
23
24
25     //#####TESTS#####
26     public static void test_all() {
27         Testing.startTests();
28         test_all_flushes();
29         test_all_two_pair();
30         test_all_pair();
31         Testing.finishTests();
32     }
33
34
35     //#####FLUSH TESTS#####
36
37     public static void test_all_flushes() {
38         Testing.startTests();
39         compare_flushes1();
40         compare_flushes2();
41         compare_flushes_tie();
42         compare_flush_2pair();
43         compare_flush_pair();
44         compare_flush_hi();
45         Testing.finishTests();
46     }
47
48     //# Flush1 vs Flush2 (Flush 1 wins highcard is greater
   )
49     public static void compare_flushes1() {

```

```

50     ArrayList<Card> handlarray = new ArrayList<Card>(
    Arrays.asList(new Card(13, "S"), new Card(12, "S"), new
    Card(9, "S"), new Card(7, "S"), new Card(3, "S")));
51     PokerHand hand1 = new PokerHand(handlarray);
52     PokerHand hand2 = new PokerHand(new ArrayList<
    Card>(Arrays.asList(new Card(4, "C"), new Card(2, "C"),
    new Card(7, "C"), new Card(5, "C"), new Card(10, "C"))));
53
54     int expected_answer = 1;
55     int actual_answer = hand1.compareTo(hand2);
56     Testing.assertEquals("Testing Flush1 vs Flush2 (
    Flush 1 wins; Highcard is greater)",
57         expected_answer,
58         actual_answer);
59 }
60
61     /// Flush1 vs Flush2 (Flush 2 wins highcard is
    greater)
62     public static void compare_flushes2() {
63         PokerHand hand1 = new PokerHand(new ArrayList<
    Card>(Arrays.asList(new Card(4, "C"), new Card(2, "C"),
    new Card(7, "C"), new Card(5, "C"), new Card(10, "C"))));
64         PokerHand hand2 = new PokerHand(new ArrayList<
    Card>(Arrays.asList(new Card(13, "S"), new Card(12, "S"
    ), new Card(9, "S"), new Card(7, "S"), new Card(3, "S"
    ))));
65
66         int expected_answer = -1;
67         int actual_answer = hand1.compareTo(hand2);
68         Testing.assertEquals("Testing Flush1 vs Flush2 (
    Flush 2 wins highcard is greater)",
69             expected_answer,
70             actual_answer);
71     }
72
73     /// Flush1 vs Flush2 (Tie)
74     public static void compare_flushes_tie() {
75         PokerHand hand1 = new PokerHand(new ArrayList<
    Card>(Arrays.asList(new Card(4, "C"), new Card(2, "C"),
    new Card(7, "C"), new Card(5, "C"), new Card(10, "C"))));
76         PokerHand hand2 = new PokerHand(new ArrayList<
    Card>(Arrays.asList(new Card(7, "S"), new Card(5, "S"),
    new Card(4, "S"), new Card(2, "S"), new Card(10, "S"))));
77
78         int expected_answer = 0;
79         int actual_answer = hand1.compareTo(hand2);
80         Testing.assertEquals("Testing Flush1 vs Flush2
    Tie",
81             expected_answer,
82             actual_answer);
83     }
84

```

```

85     /// Flush vs 2 pair
86     public static void compare_flush_2pair() {
87         PokerHand hand1 = new PokerHand(new ArrayList<
            Card>(Arrays.asList(new Card(7, "S"), new Card(5, "S"),
            new Card(11, "S"), new Card(2, "S"), new Card(10, "S"
            ))));
88         PokerHand hand2 = new PokerHand(new ArrayList<
            Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"),
            new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));
89
90         int expected_answer = 1;
91         int actual_answer = hand1.compareTo(hand2);
92         Testing.assertEquals("Testing Flush1 vs 2pair (
            Flush 1 wins)",
93             expected_answer,
94             actual_answer);
95     }
96
97     /// Flush vs pair
98     public static void compare_flush_pair() {
99         PokerHand hand1 = new PokerHand(new ArrayList<
            Card>(Arrays.asList(new Card(7, "S"), new Card(5, "S"),
            new Card(4, "S"), new Card(2, "S"), new Card(10, "S"))));
100        PokerHand hand2 = new PokerHand(new ArrayList<
            Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"),
            new Card(9, "S"), new Card(10, "C"), new Card(6, "D"))));
101
102        int expected_answer = 1;
103        int actual_answer = hand1.compareTo(hand2);
104        Testing.assertEquals("Testing Flush1 vs pair (
            Flush 1 wins)",
105            expected_answer,
106            actual_answer);
107    }
108
109    /// Flush vs high Card
110    public static void compare_flush_hi() {
111        PokerHand hand1 = new PokerHand(new ArrayList<
            Card>(Arrays.asList(new Card(7, "S"), new Card(5, "S"),
            new Card(4, "S"), new Card(2, "S"), new Card(10, "S"))));
112        PokerHand hand2 = new PokerHand(new ArrayList<
            Card>(Arrays.asList(new Card(3, "H"), new Card(4, "D"),
            new Card(10, "S"), new Card(8, "C"), new Card(6, "D"))));
113
114        int expected_answer = 1;
115        int actual_answer = hand1.compareTo(hand2);
116        Testing.assertEquals("Testing Flush1 vs highcard
            (Flush 1 wins)",
117            expected_answer,
118            actual_answer);
119    }
120

```



```

121 //#####2 PAIR TESTS#####
122
123     public static void test_all_two_pair() {
124         Testing.startTests();
125         compare_2pair_flush();
126         compare_2pair_2pair_1();
127         compare_2pair_2pair_2();
128         compare_2pair_2pair_3();
129         compare_2pair_2pair_4();
130         Testing.finishTests();
131     }
132
133     //# 2pair vs Flush
134     public static void compare_2pair_flush() {
135         PokerHand hand1 = new PokerHand(new ArrayList<
136 Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"),
137 new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));
138
139         PokerHand hand2 = new PokerHand(new ArrayList<
140 Card>(Arrays.asList(new Card(7, "S"), new Card(5, "S"),
141 new Card(11, "S"), new Card(2, "S"), new Card(10, "S"
142 ))));
143
144         int expected_answer = -1;
145         int actual_answer = hand1.compareTo(hand2);
146         Testing.assertEquals("Testing 2pair vs Flush",
147             expected_answer,
148             actual_answer);
149     }
150
151     //# 2pair1 vs 2pair2 (2pair1 wins higher of pair
152 values is greater)
153     public static void compare_2pair_2pair_1() {
154         PokerHand hand1 = new PokerHand(new ArrayList<
155 Card>(Arrays.asList(new Card(4, "H"), new Card(6, "D"),
156 new Card(10, "S"), new Card(10, "C"), new Card(4, "D"
157 ))));
158
159         PokerHand hand2 = new PokerHand(new ArrayList<
160 Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"),
161 new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));
162
163         int expected_answer = 1;
164         int actual_answer = hand1.compareTo(hand2);
165         Testing.assertEquals("Testing 2pair1 vs 2pair2 (
166 2pair1 wins higher of pair values is greater)",
167             expected_answer,
168             actual_answer);
169     }
170
171     //# 2pair1 vs 2pair2 (2pair2 wins higher of pair
172 values is greater)
173     public static void compare_2pair_2pair_2() {
174         PokerHand hand1 = new PokerHand(new ArrayList<

```

```

159 Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"),
    new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));
160     PokerHand hand2 = new PokerHand(new ArrayList<
    Card>(Arrays.asList(new Card(14, "H"), new Card(14, "D"
    ), new Card(8, "S"), new Card(8, "C"), new Card(6, "D"
    ))));
161
162     int expected_answer = -1;
163     int actual_answer = hand1.compareTo(hand2);
164     Testing.assertEquals("Testing 2pair1 vs 2pair2 (
    2pair2 wins higher of pair values is greater)",
165         expected_answer,
166         actual_answer);
167 }
168
169     ///# 2pair1 vs 2pair2 (2pair1 wins lower of pair
    values is greater)
170     public static void compare_2pair_2pair_3() {
171         PokerHand hand1 = new PokerHand(new ArrayList<
    Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"),
    new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));
172         PokerHand hand2 = new PokerHand(new ArrayList<
    Card>(Arrays.asList(new Card(3, "H"), new Card(4, "D"),
    new Card(8, "S"), new Card(8, "C"), new Card(3, "D"))));
173
174         int expected_answer = 1;
175         int actual_answer = hand1.compareTo(hand2);
176         Testing.assertEquals("2pair1 vs 2pair2 (2pair1
    wins lower of pair values is greater)",
177             expected_answer,
178             actual_answer);
179     }
180
181     ///# 2pair1 vs 2pair2 (2pair2 wins lower of pair
    values is greater)
182     public static void compare_2pair_2pair_4() {
183         PokerHand hand1 = new PokerHand(new ArrayList<
    Card>(Arrays.asList(new Card(3, "H"), new Card(4, "D"),
    new Card(8, "S"), new Card(8, "C"), new Card(3, "D"))));
184         PokerHand hand2 = new PokerHand(new ArrayList<
    Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"),
    new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));
185
186         int expected_answer = -1;
187         int actual_answer = hand1.compareTo(hand2);
188         Testing.assertEquals("Testing 2pair1 vs 2pair2 (
    2pair2 wins lower of pair values is greater)",
189             expected_answer,
190             actual_answer);
191     }
192
193     ///#####PAIR TESTS#####

```

```

194
195     public static void test_all_pair() {
196         Testing.startTests();
197         compare_pair_pair_1();
198         compare_pair_pair_2();
199         compare_pair_pair_3();
200         compare_pair_pair_4();
201         Testing.finishTests();
202     }
203
204
205     /// pair1 vs pair2 (pair1 wins; high pair)
206     public static void compare_pair_pair_1() {
207         PokerHand hand1 = new PokerHand(new ArrayList<
208 Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"),
209 new Card(9, "S"), new Card(10, "C"), new Card(6, "D"))));
210         PokerHand hand2 = new PokerHand(new ArrayList<
211 Card>(Arrays.asList(new Card(2, "H"), new Card(4, "D"),
212 new Card(9, "S"), new Card(9, "C"), new Card(6, "D"))));
213
214         int expected_answer = 1;
215         int actual_answer = hand1.compareTo(hand2);
216         Testing.assertEquals("pair1 vs pair2 (pair1 wins
217 ; high pair)",
218             expected_answer,
219             actual_answer);
220     }
221
222     /// pair1 vs pair2 (pair2 wins; high pair)
223     public static void compare_pair_pair_2() {
224         PokerHand hand1 = new PokerHand(new ArrayList<
225 Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"),
226 new Card(9, "S"), new Card(10, "C"), new Card(6, "D"))));
227         PokerHand hand2 = new PokerHand(new ArrayList<
228 Card>(Arrays.asList(new Card(12, "H"), new Card(4, "D"),
229 new Card(9, "S"), new Card(12, "C"), new Card(6, "D"))));
230
231         int expected_answer = -1;
232         int actual_answer = hand1.compareTo(hand2);
233         Testing.assertEquals("Testing pair1 vs pair2 (
234 pair2 wins; high pair)",
235             expected_answer,
236             actual_answer);
237     }
238
239     /// pair1 vs pair2 (pair1 wins; highcard)
240     public static void compare_pair_pair_3() {
241         PokerHand hand1 = new PokerHand(new ArrayList<
242 Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"),
243 new Card(12, "S"), new Card(10, "C"), new Card(6, "D"
244 ))));
245         PokerHand hand2 = new PokerHand(new ArrayList<

```

```

232 Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"),
    new Card(9, "S"), new Card(10, "C"), new Card(6, "D"))));
233
234     int expected_answer = 1;
235     int actual_answer = hand1.compareTo(hand2);
236     Testing.assertEquals("Testing pair1 vs pair2 (
pair1 wins; highcard)",
237         expected_answer,
238         actual_answer);
239 }
240
241 // # pair1 vs pair2 (pair2 wins; highcard)
242 public static void compare_pair_pair_4() {
243     PokerHand hand1 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"),
    new Card(9, "S"), new Card(10, "C"), new Card(6, "D"))));
244     PokerHand hand2 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(10, "H"), new Card(13, "D"
), new Card(9, "S"), new Card(10, "C"), new Card(6, "D"
))));
245
246     int expected_answer = -1;
247     int actual_answer = hand1.compareTo(hand2);
248     Testing.assertEquals("Testing pair1 vs pair2 (
pair2 wins; highcard",
249         expected_answer,
250         actual_answer);
251 }
252 }
253
254
255
256
257
258 /*
259
260
261
262 ( _ _ _ ) ( _ ) ( _ ) ( _ _ _ ) ( _ _ _ ) ( _ ) ( _ ) ( _ _ _ )
263 ( _ ) ( _ ) _ _ ( _ ) ( _ ) _ _ ( _ ) _ _ ( _ ) ( _ ) ( _ )
264 ( _ ) ( _ _ _ ) ( _ _ _ ) ( _ _ _ ) ( _ ) ( _ ) ( _ ) ( _ )
265 ( _ ) ( _ ) ( _ ) ( _ ) _ _ ( _ ) _ _ ( _ ) ( _ ) _ _ ( _ )
266 ( _ ) ( _ ) ( _ ) ( _ _ _ ) ( _ _ _ ) ( _ ) ( _ ) ( _ _ _ )
267
268
269 */
270
271
272
273

```

```
1 package proj4;  
2  
3 public class CommunityCardSetTests{  
4  
5 }  
6  
7
```