```java


package proj4;

import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;

/**
 * Represents a single playing card with a rank and suit.
 */
public class Card {

    private static final Map<Integer, String> RANK_NAMES = Map.ofEntries(
            Map.entry(11, "Jack"),
            Map.entry(12, "Queen"),
            Map.entry(13, "King"),
            Map.entry(14, "Ace")
    );

    private static final Map<String, Integer> STRING_TO_RANK = Map.ofEntries(
            Map.entry("two", 2),
            Map.entry("2", 2),
            Map.entry("three", 3),
            Map.entry("3", 3),
            Map.entry("four", 4),
            Map.entry("4", 4),
            Map.entry("five", 5),
            Map.entry("5", 5),
            Map.entry("six", 6),
            Map.entry("6", 6),
            Map.entry("seven", 7),
            Map.entry("7", 7),
            Map.entry("eight", 8),
            Map.entry("8", 8),
```

```
36              Map.entry("nine", 9),
37              Map.entry("9", 9),
38              Map.entry("ten", 10),
39              Map.entry("10", 10),
40              Map.entry("jack", 11),
41              Map.entry("queen", 12),
42              Map.entry("king", 13),
43              Map.entry("ace", 14)
44      );
45
46      private static final String[] SUIT_NAMES = {"Spades
    ", "Hearts", "Clubs", "Diamonds"};
47
48      private int rank;
49      private String suit;
50
51      /**
52       * constructor
53       * @param rank the rank of the card (2-14)
54       * @param suit the suit of the card (fully spelled
    out)
55       */
56      public Card(int rank, String suit) {
57          this.rank = rank;
58          this.suit = suit;
59      }
60
61      /**
62       * constructor
63       * @param rank String: whole cards (2-10) can
    either be spelled
64       * out like "two" or numeric like "2". Case
    insensitive.
65       * @param suit String: "Spades", "Hearts", "Clubs
    ", or "Diamonds"
66       */
67      public Card(String rank, String suit) {
68          String rankLower = rank.toLowerCase();
```

```
69          int rankValue = STRING_TO_RANK.get(rankLower);
70          this.rank = rankValue;
71          this.suit = suit;
72      }
73
74      /**
75       * constructor
76       * @param rank integer between 2-14
77       * @param suit integer: 0=Spades, 1=Hearts, 2=
    Clubs, or 3=Diamonds
78       */
79      public Card(int rank, int suit) {
80          this.rank = rank;
81          this.suit = SUIT_NAMES[suit];
82      }
83
84      /**
85       * Gets the rank of this card.
86       * @return the rank of the card
87       */
88      public int getRank() {
89          return this.rank;
90      }
91
92      /**
93       * Gets the suit of this card.
94       * @return the suit of the card
95       */
96      public String getSuit() {
97          return this.suit;
98      }
99
100     /**
101      * Returns a string representation of this card.
102      * @return a string representation of the card
103      */
104     public String toString() {
105         String rankStr = RANK_NAMES.getOrDefault(rank
```

```
105 , String.valueOf(rank));
106         return rankStr + " of " + suit;
107     }
108 }
```

```java
package proj4;

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

/**
 * Represents a standard 52-card deck of playing cards with shuffle and deal operations.
 */
public class Deck {

    private final int[] RANKS = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14};
    private final String[] SUITS = {"Spades", "Clubs", "Hearts", "Diamonds"};


    private ArrayList<Card> deck;
    private int nextToDeal;

    /**
     * Constructs a deck
     */
    public Deck() {
        deck = new ArrayList<>();
        for (String suit : SUITS) {
            for (int rank : RANKS) {
                deck.add(new Card(rank, suit));
            }
        }
        nextToDeal = 0;
    }


    /**
     * Shuffles the deck by swapping each card with another at random index
```

```
35      */
36     public void shuffle() {
37         for (int i = nextToDeal; i < deck.size(); i
   ++) {
38             int randomIndex = ThreadLocalRandom.current
   ().nextInt(nextToDeal, deck.size());
39             Card temp = deck.get(i);
40             deck.set(i, deck.get(randomIndex));
41             deck.set(randomIndex, temp);
42         }
43     }
44 
45     /**
46      * Checks if there are any undealt cards remaining
   in the deck.
47      *  @return false if there is else return true
48      */
49     public boolean isEmpty() {
50         return deck.size() <= nextToDeal;
51     }
52 
53     /**
54      * Returns the number of undealt cards remaining in
    the deck
55      * @return the number of undealt cards
56      */
57     public int size() {
58         return deck.size() - nextToDeal;
59     }
60 
61     /**
62      * Deals the next undealt card from the deck.
63      * Does not remove the card from the deck; instead
   tracks which cards have been dealt.
64      * @return the next undealt card, or null if there
   are no undealt cards
65      */
66     public Card deal(){
```

```
67          if (isEmpty()) {
68              return null;
69          }else{
70              Card card =  deck.get(nextToDeal);
71              nextToDeal++;
72              return card;
73          }
74      }
75
76      /**
77       * Returns all cards to an undealt state
78       */
79      public void gather(){
80          nextToDeal = 0;
81      }
82
83      /**
84       * Returns a string representation of all undealt
   cards in the deck
85       * @return a string containing all undealt cards,
   one per line
86       */
87      public String toString() {
88          String result = "";
89          for (int i = nextToDeal; i < deck.size(); i
   ++) {
90              result += deck.get(i).toString() + "\n";
91          }
92          return result;
93      }
94
95  }
```

```java
/**
 * I affirm that I have carried out the attached academic endeavors with full academic honesty, in
 * accordance with the Union College Honor Code and the course syllabus.
 * @author: James Lin
 */



package proj4;

import java.util.ArrayList;
import java.util.Scanner;

//1. Draw two new 2-card hands from a given deck. These are the hole cards.
//2. Print the community cards and the hands.
//3. Ask the user who the winner is (or if there's a tie), taking into account the community cards.
//4. If the player is correct, they get one point and the game continues.
//5. If the player is incorrect, the game ends and the player's total points should be displayed.
//6. The game is also over if there are not enough cards left in the deck to play another round.

public class Client {


    /**
     * Deals 5 community cards from the deck and creates a community card set.
     * @param deck the deck to deal from
     * @return a community card set with 5 cards
     */
    private static CommunityCardSet dealCommunityCards(Deck deck) {
```

```
30          ArrayList<Card> communityCardList = new
    ArrayList<>();
31          for (int i = 0; i < 5; i++) {
32              communityCardList.add(deck.deal());
33          }
34          return new CommunityCardSet(communityCardList);
35      }
36
37      /**
38       * Checks if the users input correctly identifies
    the winner.
39       * @param input the users input ("a", "b", or " ")
40       * @param comparison the result of handA.compareTo(
    handB)
41       * @return true if they are acutally equal, false
    otherwise
42       */
43      private static boolean isCorrectGuess(String input
    , int comparison) {
44          if (input.equals("a")) {
45              return comparison > 0;
46          } else if (input.equals("b")) {
47              return comparison < 0;
48          } else if (input.equals(" ") || input.equals(""
    )) {
49              return comparison == 0;
50          }
51          return false;
52      }
53
54      public static void main(String[] args) {
55          Scanner scanner = new Scanner(System.in);
56          Deck deck = new Deck();
57          deck.shuffle();
58
59          CommunityCardSet communityCards =
    dealCommunityCards(deck);
60
```

```
61          int score = 0;
62          boolean playing = true;
63
64          while (playing) {
65              if (deck.size() < 4) {
66                  System.out.println("Not enough cards
    left in the deck!");
67                  break;
68              }
69
70              ArrayList<Card> holeA = new ArrayList<>();
71              holeA.add(deck.deal());
72              holeA.add(deck.deal());
73              StudPokerHand handA = new StudPokerHand(
    communityCards, holeA);
74
75              ArrayList<Card> holeB = new ArrayList<>();
76              holeB.add(deck.deal());
77              holeB.add(deck.deal());
78              StudPokerHand handB = new StudPokerHand(
    communityCards, holeB);
79
80              System.out.println("The community cards are
    :");
81              System.out.println(communityCards.toString
    ());
82              System.out.println();
83              System.out.println("Which of the following
    hands is worth more?");
84              System.out.println("Hand a:");
85              System.out.println(handA.toString());
86              System.out.println("or");
87              System.out.println("Hand b:");
88              System.out.println(handB.toString());
89              System.out.println("Enter a or b (or SPACE
    to indicate they are of equal value)");
90
91              String input = scanner.nextLine();
```

```
 92              System.out.println("got input: " + input);
 93
 94              int comparison = handA.compareTo(handB);
 95              boolean correct = isCorrectGuess(input,
    comparison);
 96
 97              if (correct) {
 98                  System.out.println("CORRECT!!!");
 99                  score++;
100                  System.out.println(
    "-----------------------------------------------");
101              } else {
102                  System.out.println("INCORRECT!");
103                  playing = false;
104              }
105          }
106
107          System.out.println("Game over! Your final
    score: " + score);
108          scanner.close();
109      }
110  }
111
```

```java
package proj4;
/**
 * This class contains a collection of methods that help with testing.  All methods
 * here are static so there's no need to construct a Testing object.  Just call them
 * with the class name like so:
 * <p></p>
 * <code>Testing.assertEquals("test description", expected, actual)</code>
 *
 * @author Kristina Striegnitz, Aaron Cass, Chris Fernandes
 * @version 5/28/18
 */
public class Testing {

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

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

    /**
```

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

```
62 actual);
63         if (expected == actual) {
64             pass();
65         } else {
66             fail(message);
67         }
68     }
69 
70     public static void assertEquals(String message,
   Object expected,
71                                     Object actual)
72     {
73         String expectedString = "<<null>>";
74         String actualString = "<<null>>";
75         if (expected != null) {
76             expectedString = expected.toString();
77         }
78         if (actual != null) {
79             actualString = actual.toString();
80         }
81         printTestCaseInfo(message, expectedString,
   actualString);
82 
83         if (expected == null) {
84             if (actual == null) {
85                 pass();
86             } else {
87                 fail(message);
88             }
89         } else if (expected.equals(actual)) {
90             pass();
91         } else {
92             fail(message);
93         }
94     }
95 
96     /**
97      * Asserts that a given boolean must be true.  The
```

```
 97 test fails if
 98      * the boolean is not true.
 99      *
100      * @param message The test message
101      * @param actual The boolean value asserted to be
    true.
102      */
103     public static void assertTrue(String message,
    boolean actual)
104     {
105         assertEquals(message, true, actual);
106     }
107 
108     /**
109      * Asserts that a given boolean must be false. The
     test fails if
110      * the boolean is not false (i.e. if it is true).
111      *
112      * @param message The test message
113      * @param actual The boolean value asserted to be
    false.
114      */
115     public static void assertFalse(String message,
    boolean actual)
116     {
117         assertEquals(message, false, actual);
118     }
119 
120     private static void printTestCaseInfo(String
    message, String expected,
121                                           String
    actual)
122     {
123         if (VERBOSE) {
124             System.out.println(message + ":");
125             System.out.println("expected: " + expected
    );
126             System.out.println("actual:   " + actual);
```

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

```
163                 System.out.print("-");
164             }
165             System.out.println();
166             System.out.println();
167         }
168     }
169 
170     /**
171      * Initializes the test suite. Should be called before running any
172      * tests, so that passes and fails are correctly tallied.
173      */
174     public static void startTests()
175     {
176         System.out.println("Starting Tests");
177         System.out.println();
178         numTests = 0;
179         numFails = 0;
180     }
181 
182     /**
183      * Prints out summary data at end of tests. Should be called
184      * after all the tests have run.
185      */
186     public static void finishTests()
187     {
188         System.out.println("===============");
189         System.out.println("Tests Complete");
190         System.out.println("===============");
191         int numPasses = numTests - numFails;
192 
193         System.out.print(numPasses + "/" + numTests + " PASS ");
194         System.out.printf("(pass rate: %.1f%s)\n",
195                           100 * ((double) numPasses) / numTests,
```

```
196                             "%");
197
198         System.out.print(numFails + "/" + numTests +
    " FAIL ");
199         System.out.printf("(fail rate: %.1f%s)\n",
200                             100 * ((double) numFails) /
    numTests,
201                             "%");
202     }
203
204 }
205
```

```java
package proj4;

import java.util.*;

/**
 * Represents a poker hand of cards and can evaluate and compare it to other hands.
 */
public class PokerHand implements Comparable<PokerHand> {

    private ArrayList<Card> cards;

    private static final Map<String, Integer> CATEGORY_VALUE = Map.ofEntries(
            Map.entry("flush", 3),
            Map.entry("two pair", 2),
            Map.entry("pair", 1),
            Map.entry("high card", 0)
    );

    private static final Comparator<int[]> COMPARATOR = new Comparator<int[]>() {
        @Override
        public int compare(int[] a, int[] b) {
            if (a[0] != b[0]) {
                return b[0] - a[0];
            }
            return b[1] - a[1];
        }
    };

    private final int MAX_CARDS = 5;


    /**
     * Constructs a PokerHand with the given cards.
     * @param cardList the cards that should be in this
```

```
34  poker hand
35       */
36      public PokerHand(ArrayList<Card> cardList) {
37          this.cards = new ArrayList<>(cardList);
38      }
39
40      /**
41       * Add a card to the hand. Does nothing if the hand
    already has 5 cards.
42       * @param card the card to add
43       */
44      public void addCard(Card card) {
45          if (cards.size() < MAX_CARDS) {
46              cards.add(card);
47          }
48      }
49
50      /**
51       * Get the i-th card from the hand.
52       * @param i the index of the card to get
53       * @return the card at index i, or null if i is
   invalid
54       */
55      public Card getIthCard(int i) {
56          if (i >= 0 && i < cards.size()) {
57              return cards.get(i);
58          }
59          else{
60              return null;
61          }
62      }
63
64
65      /**
66       * Check if this hand is a flush (all cards same
   suit).
67       * @return true if the hand is a flush, else false
68       */
```

```
69      private boolean isFlush() {
70          HashSet<String> suits = new HashSet<>();
71          for (Card card : cards) {
72              suits.add(card.getSuit());
73          }
74          return suits.size() == 1;
75      }
76
77      /**
78       * Check if this hand is a straight (consecutive
    cards).
79       * @return true if the hand is a straight, else
    false
80       */
81      private boolean isStraight() {
82          ArrayList<Integer> ranks = new ArrayList<>();
83          for (Card card : cards) {
84              ranks.add(card.getRank());
85          }
86          Collections.sort(ranks);
87
88          for (int i = 0; i < 4; i++) {
89              if (ranks.get(i) + 1 != ranks.get(i + 1
    )) {
90                  return false;
91              }
92          }
93          return true;
94      }
95
96      /**
97       * Count how many cards of each rank are in this
    hand.
98       * @return a map with rank counts
99       */
100     private Map<Integer, Integer> rankCounts() {
101         Map<Integer, Integer> counts = new HashMap
    <>();
```

```
102         for (Card card : cards) {
103             int rank = card.getRank();
104             counts.put(rank, counts.getOrDefault(rank
    , 0) + 1);
105         }
106         return counts;
107     }
108 
109     /**
110      * Check if this hand has four cards of the same
    rank.
111      * @return true if the hand has four of a kind,
    else false
112      */
113     private boolean isFourOfAKind() {
114         Map<Integer, Integer> counts = rankCounts();
115         return counts.containsValue(4);
116     }
117 
118     /**
119      * Check if this hand has three of one rank and
    two of another.
120      * @return true if the hand is a full house, else
    false
121      */
122     private boolean isFullHouse() {
123         Map<Integer, Integer> counts = rankCounts();
124         boolean hasThree = counts.containsValue(3);
125         boolean hasTwo = counts.containsValue(2);
126         return hasThree && hasTwo;
127     }
128 
129     /**
130      * Check if this hand has exactly three cards of
    the same rank.
131      * @return true if the hand has three of a kind,
    else false
132      */
```

```
133     private boolean isThreeOfAKind() {
134         Map<Integer, Integer> counts = rankCounts();
135         return counts.containsValue(3) && !isFullHouse
    ();
136     }
137 
138     /**
139      * Check if this hand has exactly n pairs.
140      * Precondition: n is a positive integer and n <=
    2
141      * @param n number of pairs to check for
142      * @return true if the hand has exactly n pairs,
    else false
143      */
144     private boolean hasPairs(int n) {
145         Map<Integer, Integer> counts = rankCounts();
146         int pairCount = 0;
147         for (int count : counts.values()) {
148             if (count == 2) {
149                 pairCount++;
150             }
151         }
152         return pairCount == n;
153     }
154 
155 
156     /**
157      * Returns a string representation of this poker
    hand.
158      * @return string representation of the hand
159      */
160     public String toString() {
161         String result = "";
162         for (Card card : cards) {
163             result += "  " + card.toString() + "\n";
164         }
165         return result;
166     }
```

```
167
168     /**
169      * Evaluate this hand and return its category.
170      * @return string representing the hand category
171      */
172     private String evaluate() {
173         boolean flush = isFlush();
174         boolean straight = isStraight();
175 
176         if (flush && straight) {
177             return "flush";
178         } else if (flush) {
179             return "flush";
180         } else if (isFourOfAKind() || isFullHouse
    () || hasPairs(2)) {
181             return "two pair";
182         } else if (isThreeOfAKind() || hasPairs(1)) {
183             return "pair";
184         } else {
185             return "high card";
186         }
187     }
188 
189 
190     /**
191      * Determines how this hand compares to another
    hand, returns
192      * positive, negative, or zero depending on the
    comparison.
193      *
194      * @param other The hand to compare this hand to
195      * @return a negative number if this is worth LESS
     than other, zero
196      * if they are worth the SAME, and a positive
    number if this is worth
197      * MORE than other
198      */
199     public int compareTo(PokerHand other) {
```

```
200         String myCategory = this.evaluate();
201         int myValue = CATEGORY_VALUE.get(myCategory);
202 
203         String otherCategory = other.evaluate();
204         int otherValue = CATEGORY_VALUE.get(
    otherCategory);
205 
206         int categoryDiff = myValue - otherValue;
207         if (categoryDiff != 0) {
208             return categoryDiff;
209         }else{
210             Map<Integer, Integer> myCounts = this.
    rankCounts();
211             Map<Integer, Integer> otherCounts = other.
    rankCounts();
212 
213             List<int[]> myList = new ArrayList<>();
214             for (Map.Entry<Integer, Integer> entry :
    myCounts.entrySet()) {
215                 myList.add(new int[]{entry.getValue
    (), entry.getKey()});
216             }
217 
218             List<int[]> otherList = new ArrayList<>();
219             for (Map.Entry<Integer, Integer> entry :
    otherCounts.entrySet()) {
220                 otherList.add(new int[]{entry.getValue
    (), entry.getKey()});
221             }
222 
223             Collections.sort(myList, COMPARATOR);
224             Collections.sort(otherList, COMPARATOR);
225 
226             int minSize = Math.min(myList.size(),
    otherList.size());
227             for (int i = 0; i < minSize; i++) {
228                 int myRank = myList.get(i)[1];
229                 int otherRank = otherList.get(i)[1];
```

```
230
231                 if (myRank != otherRank) {
232                     return myRank - otherRank;
233                 }
234             }
235
236             if (myList.size() != otherList.size()) {
237                 return myList.size() - otherList.size
    ();
238             }
239
240             return 0;
241         }
242
243     }
244
245
246
247 }
```

```java
package proj4;

/**
 * Tests the Card class methods including getRank, getSuit, and toString.
 */
public class CardTester {

    /**
     * Test Card constructor and getRank method
     */
    public static void testGetRank() {
        Testing.testSection("Testing getRank");

        Card card1 = new Card(2, "Hearts");
        Testing.assertEquals("Card rank 2", 2, card1.getRank());

        Card card2 = new Card(11, "Spades");
        Testing.assertEquals("Card rank 11 (Jack)", 11, card2.getRank());

        Card card3 = new Card(14, "Diamonds");
        Testing.assertEquals("Card rank 14 (Ace)", 14, card3.getRank());
    }

    /**
     * Test Card constructor and getSuit method
     */
    public static void testGetSuit() {
        Testing.testSection("Testing getSuit");

        Card card1 = new Card(5, "Hearts");
        Testing.assertEquals("Card suit Hearts", "Hearts", card1.getSuit());

        Card card2 = new Card(10, "Clubs");
```

```
34          Testing.assertEquals("Card suit Clubs", "Clubs"
    , card2.getSuit());
35
36          Card card3 = new Card(13, "Diamonds");
37          Testing.assertEquals("Card suit Diamonds", "
    Diamonds", card3.getSuit());
38
39          Card card4 = new Card(7, "Spades");
40          Testing.assertEquals("Card suit Spades", "
    Spades", card4.getSuit());
41      }
42
43      /**
44       * Test Card toString method
45       */
46      public static void testToString() {
47          Testing.testSection("Testing toString");
48
49          Card card1 = new Card(2, "Hearts");
50          Testing.assertEquals("2 of Hearts", "2 of
    Hearts", card1.toString());
51
52          Card card2 = new Card(10, "Clubs");
53          Testing.assertEquals("10 of Clubs", "10 of
    Clubs", card2.toString());
54
55          Card card3 = new Card(11, "Spades");
56          Testing.assertEquals("Jack of Spades", "Jack of
     Spades", card3.toString());
57
58          Card card4 = new Card(12, "Diamonds");
59          Testing.assertEquals("Queen of Diamonds", "
    Queen of Diamonds", card4.toString());
60
61          Card card5 = new Card(13, "Hearts");
62          Testing.assertEquals("King of Hearts", "King of
     Hearts", card5.toString());
63
```

```
64          Card card6 = new Card(14, "Clubs");
65          Testing.assertEquals("Ace of Clubs", "Ace of
    Clubs", card6.toString());
66      }
67
68      public static void testStringConstructor() {
69          Testing.testSection("Testing String rank
    constructor");
70
71          Card card1 = new Card("two", "Hearts");
72          Testing.assertEquals("String 'two' creates
    rank 2", 2, card1.getRank());
73          Testing.assertEquals("String constructor
    toString", "2 of Hearts", card1.toString());
74
75          Card card2 = new Card("2", "Clubs");
76          Testing.assertEquals("String '2' creates rank
    2", 2, card2.getRank());
77
78          Card card3 = new Card("jack", "Spades");
79          Testing.assertEquals("String 'jack' creates
    rank 11", 11, card3.getRank());
80          Testing.assertEquals("Jack toString", "Jack of
     Spades", card3.toString());
81
82          Card card4 = new Card("QUEEN", "Diamonds");
83          Testing.assertEquals("String 'QUEEN' case
    insensitive", 12, card4.getRank());
84      }
85
86      public static void testIntSuitConstructor() {
87          Testing.testSection("Testing int suit
    constructor");
88
89          Card card1 = new Card(5, 0);
90          Testing.assertEquals("Suit 0 is Spades", "
    Spades", card1.getSuit());
91
```

```
92          Card card2 = new Card(7, 1);
93          Testing.assertEquals("Suit 1 is Hearts", "
    Hearts", card2.getSuit());
94
95          Card card3 = new Card(9, 2);
96          Testing.assertEquals("Suit 2 is Clubs", "Clubs
    ", card3.getSuit());
97
98          Card card4 = new Card(11, 3);
99          Testing.assertEquals("Suit 3 is Diamonds", "
    Diamonds", card4.getSuit());
100         Testing.assertEquals("Int suit toString", "
    Jack of Diamonds", card4.toString());
101     }
102
103
104     public static void main(String[] args) {
105         Testing.startTests();
106         testGetRank();
107         testGetSuit();
108         testToString();
109         testStringConstructor();
110         testIntSuitConstructor();
111         Testing.finishTests();
112     }
113 }
```

```java
package proj4;

/**
 * Tests the Deck class methods including shuffle, deal, size, isEmpty, gather, and toString.
 */
public class DeckTester {

    /**
     * Test Deck constructor creates 52 cards
     */
    public static void testDeckConstructor() {
        Testing.testSection("Testing Deck constructor");

        Deck deck = new Deck();
        Testing.assertEquals("New deck has 52 cards", 52, deck.size());
        Testing.assertFalse("New deck is not empty", deck.isEmpty());
    }

    /**
     * Test Deck size method
     */
    public static void testSize() {
        Testing.testSection("Testing size");

        Deck deck = new Deck();
        Testing.assertEquals("Initial size is 52", 52, deck.size());

        deck.deal();
        Testing.assertEquals("Size after dealing one card is 51", 51, deck.size());

        for (int i = 0; i < 5; i++) {
            deck.deal();
```

```
33          }
34          Testing.assertEquals("Size after dealing 6
    cards total is 46", 46, deck.size());
35      }
36
37      /**
38       * Test Deck isEmpty method
39       */
40      public static void testIsEmpty() {
41          Testing.testSection("Testing isEmpty");
42          Deck deck = new Deck();
43          Testing.assertFalse("New deck is not empty",
    deck.isEmpty());
44
45          for (int i = 0; i < 52; i++) {
46              deck.deal();
47          }
48          Testing.assertTrue("Deck is empty after dealing
     all cards", deck.isEmpty());
49          Card nullCard = deck.deal();
50          Testing.assertEquals("Dealing from empty deck
    returns null", null, nullCard);
51      }
52
53      /**
54       * Test Deck deal method
55       */
56      public static void testDeal() {
57          Testing.testSection("Testing deal");
58
59          Deck deck = new Deck();
60
61          Card card1 = deck.deal();
62          Testing.assertEquals("First card rank is 2", 2
    , card1.getRank());
63          Testing.assertEquals("First card suit is Spades
    ", "Spades", card1.getSuit());
64
```

```
65          Testing.assertEquals("Size after one deal is
    51", 51, deck.size());
66
67          Card card2 = deck.deal();
68          Testing.assertFalse("Second card is different
    from first", card1.getRank() == card2.getRank() &&
    card1.getSuit().equals(card2.getSuit()));
69      }
70
71      /**
72       * Test Deck gather method
73       */
74      public static void testGather() {
75          Testing.testSection("Testing gather");
76
77          Deck deck = new Deck();
78          for (int i = 0; i < 10; i++) {
79              deck.deal();
80          }
81          Testing.assertEquals("Size after dealing 10
    cards is 42", 42, deck.size());
82
83          deck.gather();
84          Testing.assertEquals("Size after gather is 52"
    , 52, deck.size());
85          Testing.assertFalse("Deck is not empty after
    gather", deck.isEmpty());
86      }
87
88      /**
89       * Test Deck shuffle method
90       */
91      public static void testShuffle() {
92          Testing.testSection("Testing shuffle");
93
94          Deck deck1 = new Deck();
95          Deck deck2 = new Deck();
96
```

```
 97          Card[] unshuffled = new Card[5];
 98          for (int i = 0; i < 5; i++) {
 99              unshuffled[i] = deck1.deal();
100          }
101
102          deck2.shuffle();
103          Card[] shuffled = new Card[5];
104          for (int i = 0; i < 5; i++) {
105              shuffled[i] = deck2.deal();
106          }
107
108          deck1.gather();
109          deck1.shuffle();
110          Testing.assertEquals("Shuffled deck still has
    52 cards", 52, deck1.size());
111
112          Card card = deck1.deal();
113          Testing.assertFalse("Can still deal after
    shuffle", card == null);
114      }
115
116      /**
117       * Test Deck toString method
118       */
119      public static void testToString() {
120          Testing.testSection("Testing toString");
121
122          Deck deck = new Deck();
123          String deckStr = deck.toString();
124          Testing.assertFalse("toString returns non-
    empty string", deckStr.isEmpty());
125          Testing.assertTrue("toString contains card
    descriptions", deckStr.contains("of"));
126          for (int i = 0; i < 10; i++) {
127              deck.deal();
128          }
129          String shorterStr = deck.toString();
130          Testing.assertTrue("toString is shorter after
```

```
130 dealing cards", shorterStr.length() < deckStr.length
    ());
131     }
132
133
134     public static void main(String[] args) {
135         Testing.startTests();
136         testDeckConstructor();
137         testSize();
138         testIsEmpty();
139         testDeal();
140         testGather();
141         testShuffle();
142         testToString();
143         Testing.finishTests();
144     }
145 }
146
```

```java
package proj4;

import java.util.ArrayList;

/**
 * Represents a 2-card poker hand with access to community cards.
 */
public class StudPokerHand{

    private final int HOLECARDS_NUM = 2;
    private final int TOTAL_CARDS = 7;
    private final int POKER_HAND_SIZE = 5;
    private ArrayList<Card> holeCards;
    private CommunityCardSet communityCardSets;

    /**
     * Constructs a stud poker hand with the given community cards and hole cards
     * @param cc the community card set
     * @param cardList the hole cards for this hand
     */
    public StudPokerHand(CommunityCardSet cc, ArrayList<Card> cardList) {
        this.communityCardSets = cc;
        this.holeCards = cardList;
    }

    /**
     * Adds a card to the hole cards. Does nothing if the hand already has 2 cards.
     * @param card the card to add
     */
    public void addCard(Card card) {
        if (holeCards.size() < HOLECARDS_NUM) {
            holeCards.add(card);
        }
    }
```

```
35
36      /**
37       * Gets the i-th card from the hole cards.
38       * @param i the index of the card to get
39       * @return the card at index i, or null if i is
   invalid
40       */
41      public Card getIthCard(int i) {
42          if (i >= 0 && i < holeCards.size()) {
43              return holeCards.get(i);
44          }else{
45              return null;
46          }
47      }
48
49      /**
50       * Gets all seven cards in the hand, including the
   hole cards and community cards.
51       * @return a list of all cards in the hand
52       */
53      private ArrayList<Card> getAllSevenCards() {
54          ArrayList<Card> allCards = new ArrayList<>();
55
56          for (int i = 0; i < holeCards.size(); i++) {
57              allCards.add(holeCards.get(i));
58          }
59
60          for (int i = 0; i < CommunityCardSet.
   MAX_CARD_NUM; i++) {
61              Card c = communityCardSets.getIthCard(i);
62              if (c != null) {
63                  allCards.add(c);
64              }
65          }
66
67          return allCards;
68      }
69
```

```
70      /**
71       * Gets all possible five card hands from the
    seven cards in the hand.
72       * @return a list of all possible five card hands
73       */
74      private ArrayList<PokerHand> getAllFiveCardHands
    () {
75          ArrayList<Card> sevenCards = getAllSevenCards
    ();
76          ArrayList<PokerHand> hands = new ArrayList
    <>();
77          int numCards = sevenCards.size();
78
79          for (int i = 0; i < numCards; i++) {
80              for (int j = i + 1; j < numCards; j++) {
81                  ArrayList<Card> fiveCards = new
    ArrayList<>();
82                  for (int k = 0; k < numCards; k++) {
83                      fiveCards.add(sevenCards.get(k));
84                  }
85                  fiveCards.remove(j);
86                  fiveCards.remove(i);
87                  hands.add(new PokerHand(fiveCards));
88              }
89          }
90
91          return hands;
92      }
93
94      /**
95       * Gets the best five card hand from the seven
    cards in the hand.
96       * @return the best five card hand
97       */
98      private PokerHand getBestFiveCardHand()
99      {
100         ArrayList<PokerHand> hands =
    getAllFiveCardHands();
```

```
101         PokerHand bestSoFar = hands.get(0);
102         for (int i = 1; i < hands.size(); i++) {
103             if (hands.get(i).compareTo(bestSoFar) > 0
    ) {
104                 bestSoFar = hands.get(i);
105             }
106         }
107         return bestSoFar;
108     }
109 
110 
111     /**
112      * Returns the hole cards separated by comma and
    space.
113      * Example: "4 of Clubs, 8 of Spades"
114      */
115     public String toString() {
116         if (holeCards == null || holeCards.isEmpty
    ()) {
117             return "";
118         }
119         String result = "";
120         for (int i = 0; i < holeCards.size(); i++) {
121             if (i > 0) {
122                 result += ", ";
123             }
124             result += holeCards.get(i).toString();
125         }
126         return result;
127     }
128 
129     /**
130      * Determines how this hand compares to another
    hand, using the
131      * community card set to determine the best 5-card
     hand it can
132      * make. Returns positive, negative, or zero
    depending on the comparison.
```

```
133      *
134      * @param other The hand to compare this hand to
135      * @return a negative number if this is worth LESS
     than other, zero
136      * if they are worth the SAME, and a positive
    number if this is worth
137      * MORE than other
138      */
139     public int compareTo(StudPokerHand other) {
140         if (other == null) {
141             return -1;
142         }
143         PokerHand thisBest = this.getBestFiveCardHand
    ();
144         PokerHand otherBest = other.
    getBestFiveCardHand();
145 
146         return thisBest.compareTo(otherBest);
147     }
148 
149 
150 
151 }
```

```java
package proj4;

import java.util.*;

/**
 * Represents a set of community cards shared by all players in a poker game.
 */
public class CommunityCardSet {

    public static final int MAX_CARD_NUM = 5;
    private ArrayList<Card> cards;

    /**
     * Constructs a community card set with the given cards
     * @param cards the cards in the community set
     */
    public CommunityCardSet(ArrayList<Card> cards) {
        this.cards = cards;
    }

    /**
     * Adds a card to the community set. Does nothing if the set already has 5 cards.
     * @param card the card to add
     */
    public void addCard(Card card) {
        if (cards.size() < MAX_CARD_NUM) {
            cards.add(card);
        }
    }

    /**
     * Gets the i-th card from the community set.
     * @param i the index of the card to get
     * @return the card at index i, or null if i is invalid
```

```
35      */
36     public Card getIthCard(int i) {
37         if (i >= 0 && i < cards.size()) {
38             return cards.get(i);
39         }
40         return null;
41     }
42 
43     /**
44      * Returns the community cards as a single line separated by  | .
45      * Example: "King of Spades | 6 of Clubs | 7 of Diamonds"
46      */
47     public String toString() {
48         if (cards == null || cards.isEmpty()) {
49             return "";
50         }
51         String result = "";
52         for (int i = 0; i < cards.size(); i++) {
53             if (i > 0) {
54                 result += " | ";
55             }
56             result += cards.get(i).toString();
57         }
58         return result;
59     }
60 
61 }
62 
```

```java
package proj4;

import java.util.ArrayList;

public class StudPokerHandTester {

    public static void testConstructor() {
        Testing.testSection("Testing StudPokerHand constructor");

        ArrayList<Card> communityCards = new ArrayList<>();
        communityCards.add(new Card(2, "Hearts"));
        communityCards.add(new Card(5, "Spades"));
        communityCards.add(new Card(9, "Clubs"));
        communityCards.add(new Card(11, "Diamonds"));
        communityCards.add(new Card(14, "Hearts"));
        CommunityCardSet ccs = new CommunityCardSet(communityCards);

        ArrayList<Card> holeCards = new ArrayList<>();
        holeCards.add(new Card(3, "Hearts"));
        holeCards.add(new Card(7, "Spades"));
        StudPokerHand hand = new StudPokerHand(ccs, holeCards);

        Testing.assertEquals("First hole card is 3 of Hearts", "3 of Hearts", hand.getIthCard(0).toString());
        Testing.assertEquals("Second hole card is 7 of Spades", "7 of Spades", hand.getIthCard(1).toString());
    }

    public static void testAddCard() {
        Testing.testSection("Testing addCard");

        ArrayList<Card> communityCards = new ArrayList<>();
        communityCards.add(new Card(2, "Hearts"));
```

```
32          communityCards.add(new Card(5, "Spades"));
33          communityCards.add(new Card(9, "Clubs"));
34          communityCards.add(new Card(11, "Diamonds"));
35          communityCards.add(new Card(14, "Hearts"));
36          CommunityCardSet ccs = new CommunityCardSet(
    communityCards);
37
38          ArrayList<Card> holeCards = new ArrayList<>();
39          holeCards.add(new Card(3, "Hearts"));
40          StudPokerHand hand = new StudPokerHand(ccs,
    holeCards);
41
42          hand.addCard(new Card(7, "Spades"));
43          Testing.assertEquals("Second card added", "7 of
     Spades", hand.getIthCard(1).toString());
44
45          hand.addCard(new Card(10, "Diamonds"));
46          Testing.assertEquals("Third card not added",
    null, hand.getIthCard(2));
47      }
48
49      public static void testGetIthCard() {
50          Testing.testSection("Testing getIthCard");
51
52          ArrayList<Card> communityCards = new ArrayList
    <>();
53          communityCards.add(new Card(2, "Hearts"));
54          communityCards.add(new Card(5, "Spades"));
55          communityCards.add(new Card(9, "Clubs"));
56          communityCards.add(new Card(11, "Diamonds"));
57          communityCards.add(new Card(14, "Hearts"));
58          CommunityCardSet ccs = new CommunityCardSet(
    communityCards);
59
60          ArrayList<Card> holeCards = new ArrayList<>();
61          holeCards.add(new Card(6, "Clubs"));
62          holeCards.add(new Card(8, "Diamonds"));
63          StudPokerHand hand = new StudPokerHand(ccs,
```

```
63 holeCards);
64
65         Testing.assertEquals("Get first card", "6 of
   Clubs", hand.getIthCard(0).toString());
66         Testing.assertEquals("Get second card", "8 of
   Diamonds", hand.getIthCard(1).toString());
67         Testing.assertEquals("Invalid index returns
   null", null, hand.getIthCard(2));
68         Testing.assertEquals("Negative index returns
   null", null, hand.getIthCard(-1));
69     }
70
71     public static void testToString() {
72         Testing.testSection("Testing toString");
73
74         ArrayList<Card> communityCards = new ArrayList
   <>();
75         communityCards.add(new Card(2, "Hearts"));
76         communityCards.add(new Card(5, "Spades"));
77         communityCards.add(new Card(9, "Clubs"));
78         communityCards.add(new Card(11, "Diamonds"));
79         communityCards.add(new Card(14, "Hearts"));
80         CommunityCardSet ccs = new CommunityCardSet(
   communityCards);
81
82         ArrayList<Card> holeCards = new ArrayList<>();
83         holeCards.add(new Card(6, "Spades"));
84         holeCards.add(new Card(8, "Clubs"));
85         StudPokerHand hand = new StudPokerHand(ccs,
   holeCards);
86
87         String expected = "6 of Spades, 8 of Clubs";
88         Testing.assertEquals("toString format",
   expected, hand.toString());
89     }
90
91     public static void testCompareTo() {
92         Testing.testSection("Testing compareTo");
```

```
 93
 94          ArrayList<Card> communityCards = new ArrayList
    <>();
 95          communityCards.add(new Card(12, "Spades"));
 96          communityCards.add(new Card(5, "Diamonds"));
 97          communityCards.add(new Card(2, "Spades"));
 98          communityCards.add(new Card(6, "Clubs"));
 99          communityCards.add(new Card(7, "Diamonds"));
100          CommunityCardSet ccs = new CommunityCardSet(
    communityCards);
101
102          ArrayList<Card> holeA = new ArrayList<>();
103          holeA.add(new Card(6, "Spades"));
104          holeA.add(new Card(8, "Clubs"));
105          StudPokerHand handA = new StudPokerHand(ccs,
    holeA);
106
107          ArrayList<Card> holeB = new ArrayList<>();
108          holeB.add(new Card(4, "Clubs"));
109          holeB.add(new Card(8, "Spades"));
110          StudPokerHand handB = new StudPokerHand(ccs,
    holeB);
111
112          Testing.assertTrue("Hand A is better than Hand
     B", handA.compareTo(handB) > 0);
113
114          ArrayList<Card> holeC = new ArrayList<>();
115          holeC.add(new Card(13, "Hearts"));
116          holeC.add(new Card(2, "Hearts"));
117          StudPokerHand handC = new StudPokerHand(ccs,
    holeC);
118
119          ArrayList<Card> holeD = new ArrayList<>();
120          holeD.add(new Card(6, "Hearts"));
121          holeD.add(new Card(11, "Spades"));
122          StudPokerHand handD = new StudPokerHand(ccs,
    holeD);
123
```

```
124         Testing.assertTrue("Hand D is better than Hand
     C", handD.compareTo(handC) > 0);
125     }
126 
127     public static void testFlushBeatsAces() {
128         Testing.testSection("Testing flush beats pair
    of aces");
129 
130         ArrayList<Card> communityCards = new ArrayList
    <>();
131         communityCards.add(new Card(2, "Hearts"));
132         communityCards.add(new Card(5, "Hearts"));
133         communityCards.add(new Card(8, "Hearts"));
134         communityCards.add(new Card(10, "Hearts"));
135         communityCards.add(new Card(3, "Clubs"));
136         CommunityCardSet ccs = new CommunityCardSet(
    communityCards);
137 
138         ArrayList<Card> holeA = new ArrayList<>();
139         holeA.add(new Card(14, "Spades"));
140         holeA.add(new Card(14, "Diamonds"));
141         StudPokerHand handA = new StudPokerHand(ccs,
    holeA);
142 
143         ArrayList<Card> holeB = new ArrayList<>();
144         holeB.add(new Card(7, "Hearts"));
145         holeB.add(new Card(4, "Clubs"));
146         StudPokerHand handB = new StudPokerHand(ccs,
    holeB);
147 
148         Testing.assertTrue("Hand B (flush) beats Hand
    A (pair of aces)", handB.compareTo(handA) > 0);
149     }
150 
151     public static void main(String[] args) {
152         Testing.startTests();
153         testConstructor();
154         testAddCard();
```

```
155         testGetIthCard();
156         testToString();
157         testCompareTo();
158         testFlushBeatsAces();
159         Testing.finishTests();
160     }
161 }
162
```

```java
package proj4;

import java.util.ArrayList;

/**
 * Tests the PokerHand comparison logic for different hand categories and tie scenarios.
 *
 * I affirm that I have carried out the attached academic endeavors with full academic honesty,
 * in accordance with the Union College Honor Code and the course syllabus.
 * @author James Lin
 */
public class PokerComparisonTests {

    /**
     * Test that flush beats high card
     */
    public static void flushVsHighCard() {
        String mess = "Flush vs high card";
        ArrayList<Card> flushCards = new ArrayList<>();
        flushCards.add(new Card(2, "Hearts"));
        flushCards.add(new Card(5, "Hearts"));
        flushCards.add(new Card(9, "Hearts"));
        flushCards.add(new Card(13, "Hearts"));
        flushCards.add(new Card(14, "Hearts"));
        PokerHand flush = new PokerHand(flushCards);

        ArrayList<Card> highCardCards = new ArrayList<>();
        highCardCards.add(new Card(2, "Diamonds"));
        highCardCards.add(new Card(5, "Clubs"));
        highCardCards.add(new Card(7, "Spades"));
        highCardCards.add(new Card(9, "Diamonds"));
        highCardCards.add(new Card(11, "Clubs"));
        PokerHand highCard = new PokerHand(highCardCards);
```

```
34
35          int actual = flush.compareTo(highCard);
36          Testing.assertTrue(mess, actual > 0);
37      }
38
39      /**
40       * Test that flush beats pair
41       */
42      public static void flushVsPair() {
43          String mess = "Flush vs pair";
44          ArrayList<Card> flushCards = new ArrayList<>();
45          flushCards.add(new Card(2, "Diamonds"));
46          flushCards.add(new Card(4, "Diamonds"));
47          flushCards.add(new Card(6, "Diamonds"));
48          flushCards.add(new Card(8, "Diamonds"));
49          flushCards.add(new Card(10, "Diamonds"));
50          PokerHand flush = new PokerHand(flushCards);
51
52          ArrayList<Card> pairCards = new ArrayList<>();
53          pairCards.add(new Card(14, "Hearts"));
54          pairCards.add(new Card(14, "Diamonds"));
55          pairCards.add(new Card(13, "Clubs"));
56          pairCards.add(new Card(12, "Spades"));
57          pairCards.add(new Card(11, "Hearts"));
58          PokerHand pair = new PokerHand(pairCards);
59
60          int actual = flush.compareTo(pair);
61          Testing.assertTrue(mess, actual > 0);
62      }
63
64      /**
65       * Test comparing two high card hands
66       */
67      public static void highCardVsHighCard() {
68          String mess = "High card vs high card";
69          ArrayList<Card> kingHighCards = new ArrayList
    <>();
70          kingHighCards.add(new Card(2, "Hearts"));
```

```
71          kingHighCards.add(new Card(4, "Diamonds"));
72          kingHighCards.add(new Card(7, "Clubs"));
73          kingHighCards.add(new Card(9, "Spades"));
74          kingHighCards.add(new Card(13, "Hearts"));
75          PokerHand kingHigh = new PokerHand(
    kingHighCards);
76
77          ArrayList<Card> queenHighCards = new ArrayList
    <>();
78          queenHighCards.add(new Card(2, "Diamonds"));
79          queenHighCards.add(new Card(4, "Clubs"));
80          queenHighCards.add(new Card(7, "Spades"));
81          queenHighCards.add(new Card(9, "Diamonds"));
82          queenHighCards.add(new Card(12, "Clubs"));
83          PokerHand queenHigh = new PokerHand(
    queenHighCards);
84
85          int actual = kingHigh.compareTo(queenHigh);
86          Testing.assertTrue(mess, actual > 0);
87      }
88
89      /**
90       * Test that same ranks but different suits are a
    tie
91       */
92      public static void identicalHandsTie() {
93          String mess = "Same ranks but different suits"
    ;
94          ArrayList<Card> hand1Cards = new ArrayList
    <>();
95          hand1Cards.add(new Card(2, "Hearts"));
96          hand1Cards.add(new Card(5, "Diamonds"));
97          hand1Cards.add(new Card(9, "Clubs"));
98          hand1Cards.add(new Card(13, "Spades"));
99          hand1Cards.add(new Card(14, "Hearts"));
100         PokerHand hand1 = new PokerHand(hand1Cards);
101
102         ArrayList<Card> hand2Cards = new ArrayList
```

```
102 <>();
103         hand2Cards.add(new Card(2, "Diamonds"));
104         hand2Cards.add(new Card(5, "Clubs"));
105         hand2Cards.add(new Card(9, "Hearts"));
106         hand2Cards.add(new Card(13, "Diamonds"));
107         hand2Cards.add(new Card(14, "Clubs"));
108         PokerHand hand2 = new PokerHand(hand2Cards);
109 
110         int actual = hand1.compareTo(hand2);
111         Testing.assertEquals(mess, 0, actual);
112     }
113 
114     /**
115      * Test that same pair with same other cards are a
    tie
116      */
117     public static void samePairSameOtherCardsTie() {
118         String mess = "Same pair, same other cards";
119         ArrayList<Card> hand1Cards = new ArrayList
    <>();
120         hand1Cards.add(new Card(10, "Hearts"));
121         hand1Cards.add(new Card(10, "Diamonds"));
122         hand1Cards.add(new Card(5, "Clubs"));
123         hand1Cards.add(new Card(3, "Spades"));
124         hand1Cards.add(new Card(2, "Hearts"));
125         PokerHand hand1 = new PokerHand(hand1Cards);
126 
127         ArrayList<Card> hand2Cards = new ArrayList
    <>();
128         hand2Cards.add(new Card(10, "Clubs"));
129         hand2Cards.add(new Card(10, "Spades"));
130         hand2Cards.add(new Card(5, "Diamonds"));
131         hand2Cards.add(new Card(3, "Hearts"));
132         hand2Cards.add(new Card(2, "Clubs"));
133         PokerHand hand2 = new PokerHand(hand2Cards);
134 
135         int actual = hand1.compareTo(hand2);
136         Testing.assertEquals(mess, 0, actual);
```

```
137     }
138
139     /**
140      * Test comparing two two pair hands
141      */
142     public static void twoPairVsTwoPair() {
143         String mess = "Two pair vs two pair";
144         ArrayList<Card> twoPair1Cards = new ArrayList
    <>();
145         twoPair1Cards.add(new Card(9, "Hearts"));
146         twoPair1Cards.add(new Card(9, "Diamonds"));
147         twoPair1Cards.add(new Card(8, "Clubs"));
148         twoPair1Cards.add(new Card(8, "Spades"));
149         twoPair1Cards.add(new Card(3, "Hearts"));
150         PokerHand twoPair1 = new PokerHand(
    twoPair1Cards);
151
152         ArrayList<Card> twoPair2Cards = new ArrayList
    <>();
153         twoPair2Cards.add(new Card(9, "Clubs"));
154         twoPair2Cards.add(new Card(9, "Spades"));
155         twoPair2Cards.add(new Card(7, "Diamonds"));
156         twoPair2Cards.add(new Card(7, "Hearts"));
157         twoPair2Cards.add(new Card(3, "Clubs"));
158         PokerHand twoPair2 = new PokerHand(
    twoPair2Cards);
159
160         int actual = twoPair1.compareTo(twoPair2);
161         Testing.assertTrue(mess, actual > 0);
162     }
163
164     /**
165      * Test that full house beats two pair
166      */
167     public static void fullHouseVsTwoPair() {
168         String mess = "Full house vs two pair";
169         ArrayList<Card> fullHouseCards = new ArrayList
    <>();
```

```
170         fullHouseCards.add(new Card(5, "Hearts"));
171         fullHouseCards.add(new Card(5, "Diamonds"));
172         fullHouseCards.add(new Card(5, "Clubs"));
173         fullHouseCards.add(new Card(7, "Spades"));
174         fullHouseCards.add(new Card(7, "Hearts"));
175         PokerHand fullHouse = new PokerHand(
    fullHouseCards);
176 
177         ArrayList<Card> twoPairCards = new ArrayList
    <>();
178         twoPairCards.add(new Card(7, "Diamonds"));
179         twoPairCards.add(new Card(7, "Clubs"));
180         twoPairCards.add(new Card(6, "Spades"));
181         twoPairCards.add(new Card(6, "Hearts"));
182         twoPairCards.add(new Card(3, "Diamonds"));
183         PokerHand twoPair = new PokerHand(twoPairCards
    );
184 
185         int actual = fullHouse.compareTo(twoPair);
186         Testing.assertTrue(mess, actual < 0);
187     }
188 
189     /**
190      * Test four of a kind vs full house
191      */
192     public static void fourKindVsFullHouse() {
193         String mess = "Four of a kind vs full house";
194         ArrayList<Card> fourKindCards = new ArrayList
    <>();
195         fourKindCards.add(new Card(13, "Hearts"));
196         fourKindCards.add(new Card(13, "Diamonds"));
197         fourKindCards.add(new Card(13, "Clubs"));
198         fourKindCards.add(new Card(13, "Spades"));
199         fourKindCards.add(new Card(2, "Hearts"));
200         PokerHand fourKind = new PokerHand(
    fourKindCards);
201 
202         ArrayList<Card> fullHouseCards = new ArrayList
```

```
202 <>();
203         fullHouseCards.add(new Card(14, "Hearts"));
204         fullHouseCards.add(new Card(14, "Diamonds"));
205         fullHouseCards.add(new Card(14, "Clubs"));
206         fullHouseCards.add(new Card(3, "Spades"));
207         fullHouseCards.add(new Card(3, "Hearts"));
208         PokerHand fullHouse = new PokerHand(
    fullHouseCards);
209 
210         int actual = fourKind.compareTo(fullHouse);
211         Testing.assertTrue(mess, actual < 0);
212     }
213 
214     /**
215      * Test comparing hands from different categories
216      */
217     public static void testDifferentCategories() {
218         Testing.testSection("Testing different
    categories");
219         flushVsHighCard();
220         flushVsPair();
221     }
222 
223     /**
224      * Test comparing hands within the same category
225      */
226     public static void testSameCategory() {
227         Testing.testSection("Testing same category");
228         highCardVsHighCard();
229         twoPairVsTwoPair();
230     }
231 
232     /**
233      * Test hands that should tie
234      */
235     public static void testTies() {
236         Testing.testSection("Testing ties");
237         identicalHandsTie();
```

```
238         samePairSameOtherCardsTie();
239     }
240 
241     /**
242      * Test complex cases within same category
243      */
244     public static void testComplexSameCategory() {
245         Testing.testSection("Testing complex same
    category");
246         fullHouseVsTwoPair();
247         fourKindVsFullHouse();
248     }
249 
250 
251     public static void main(String[] args) {
252         Testing.startTests();
253         testDifferentCategories();
254         testSameCategory();
255         testTies();
256         testComplexSameCategory();
257         Testing.finishTests();
258     }
259 }
260 
```

```java
package proj4;

import java.util.ArrayList;

public class CommunityCardSetTester {

    public static void testConstructor() {
        Testing.testSection("Testing CommunityCardSet constructor");

        ArrayList<Card> cards = new ArrayList<>();
        cards.add(new Card(2, "Hearts"));
        cards.add(new Card(10, "Spades"));
        cards.add(new Card(14, "Diamonds"));

        CommunityCardSet ccs = new CommunityCardSet(cards);
        Testing.assertEquals("First card is 2 of Hearts", "2 of Hearts", ccs.getIthCard(0).toString());
        Testing.assertEquals("Second card is 10 of Spades", "10 of Spades", ccs.getIthCard(1).toString());
        Testing.assertEquals("Third card is Ace of Diamonds", "Ace of Diamonds", ccs.getIthCard(2).toString());
    }


    public static void testAddCard() {
        Testing.testSection("Testing addCard");

        ArrayList<Card> cards = new ArrayList<>();
        CommunityCardSet ccs = new CommunityCardSet(cards);

        ccs.addCard(new Card(5, "Clubs"));
        Testing.assertEquals("First card added", "5 of Clubs", ccs.getIthCard(0).toString());

```

```
31          ccs.addCard(new Card(11, "Hearts"));
32          ccs.addCard(new Card(2, "Spades"));
33          ccs.addCard(new Card(3, "Spades"));
34          ccs.addCard(new Card(4, "Spades"));
35          Testing.assertEquals("Fifth card added", "4 of
    Spades", ccs.getIthCard(4).toString());
36
37          ccs.addCard(new Card(10, "Diamonds"));
38          Testing.assertEquals("Sixth card not added",
    null, ccs.getIthCard(5));
39      }
40
41      public static void testGetIthCard() {
42          Testing.testSection("Testing getIthCard");
43
44          ArrayList<Card> cards = new ArrayList<>();
45          cards.add(new Card(7, "Hearts"));
46          cards.add(new Card(9, "Clubs"));
47          cards.add(new Card(13, "Spades"));
48
49          CommunityCardSet ccs = new CommunityCardSet(
    cards);
50
51          Testing.assertEquals("Get card at index 0", "7
    of Hearts", ccs.getIthCard(0).toString());
52          Testing.assertEquals("Get card at index 1", "9
    of Clubs", ccs.getIthCard(1).toString());
53          Testing.assertEquals("Get card at index 2", "
    King of Spades", ccs.getIthCard(2).toString());
54          Testing.assertEquals("Invalid index returns
    null", null, ccs.getIthCard(3));
55          Testing.assertEquals("Negative index returns
    null", null, ccs.getIthCard(-1));
56      }
57
58      public static void testToString() {
59          Testing.testSection("Testing toString");
60
```

```
61          ArrayList<Card> cards = new ArrayList<>();
62          cards.add(new Card(12, "Spades"));
63          cards.add(new Card(5, "Diamonds"));
64          cards.add(new Card(2, "Spades"));
65          cards.add(new Card(6, "Clubs"));
66          cards.add(new Card(7, "Diamonds"));
67
68          CommunityCardSet ccs = new CommunityCardSet(
    cards);
69          String expected = "Queen of Spades | 5 of
    Diamonds | 2 of Spades | 6 of Clubs | 7 of Diamonds";
70          Testing.assertEquals("toString format",
    expected, ccs.toString());
71
72          ArrayList<Card> oneCard = new ArrayList<>();
73          oneCard.add(new Card(14, "Hearts"));
74          CommunityCardSet ccs2 = new CommunityCardSet(
    oneCard);
75          Testing.assertEquals("toString with one card",
    "Ace of Hearts", ccs2.toString());
76
77          ArrayList<Card> empty = new ArrayList<>();
78          CommunityCardSet ccs3 = new CommunityCardSet(
    empty);
79          Testing.assertEquals("toString with empty set"
    , "", ccs3.toString());
80      }
81
82      public static void main(String[] args) {
83          Testing.startTests();
84          testConstructor();
85          testAddCard();
86          testGetIthCard();
87          testToString();
88          Testing.finishTests();
89      }
90  }
```