```java
package proj3; // do not erase. Gradescope expects this.

import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;
// Constructor, getRank, getSuit, __str__

/**
 * 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 int rank;
    private String suit;

    /**
     * Constructs a Card with the specified rank and suit.
     * @param rank the rank of the card (2-14)
     * @param suit the suit of the card (fully spelled out)
     */
    public Card(int rank, String suit) {
        this.rank = rank;
        this.suit = suit;
    }

    /**
```

```
34      * Gets the rank of this card.
35      * @return the rank of the card
36      */
37     public int getRank() {
38         return this.rank;
39     }
40
41     /**
42      * Gets the suit of this card.
43      * @return the suit of the card
44      */
45     public String getSuit() {
46         return this.suit;
47     }
48
49     /**
50      * Returns a string representation of this card.
51      * @return a string representation of the card
52      */
53     public String toString() {
54         String rankStr = RANK_NAMES.getOrDefault(rank,
   String.valueOf(rank));
55         return rankStr + " of " + suit;
56     }
57 }
58
```

```java
package proj3; // do not erase. Gradescope expects this.

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
```

```
34 another at random index
35      */
36     public void shuffle() {
37         for (int i = 0; i < deck.size(); i++) {
38             int randomIndex = ThreadLocalRandom.current
   ().nextInt(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()) return null;
68
69          Card card =  deck.get(nextToDeal);
70          nextToDeal++;
71          return card;
72      }
73
74      /**
75       * Returns all cards to an undealt state
76       */
77      public void gather(){
78          nextToDeal = 0;
79      }
80
81      /**
82       * Returns a string representation of all undealt
   cards in the deck
83       * @return a string containing all undealt cards,
   one per line
84       */
85      public String toString() {
86          String result = "";
87          for (int i = nextToDeal; i < deck.size(); i
   ++) {
88              result += deck.get(i).toString() + "\n";
89          }
90          return result;
91      }
92
93  }
94
```

```java
package proj3;
import java.util.ArrayList;
import java.util.Scanner;

/**
 * A poker game client where players guess which hand wins in multiple rounds.
 */
public class Client {

    /**
     * Deals a single 5 card poker hand from the deck.
     * @param deck the deck to deal from
     * @return a PokerHand containing 5 cards
     */
    public static PokerHand dealHand(Deck deck) {
        ArrayList<Card> cards = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            cards.add(deck.deal());
        }
        return new PokerHand(cards);
    }

    /**
     * Runs the poker guessing game where players predict which hand wins.
     */
    public static void main(String[] args) {
        Deck deck = new Deck();
        deck.shuffle();

        Scanner scanner = new Scanner(System.in);
        int roundNumber = 1;
        int correctGuesses = 0;

        while (deck.size() >= 10) {
            System.out.println("Round " + roundNumber);

```

```
37              PokerHand hand1 = dealHand(deck);
38              PokerHand hand2 = dealHand(deck);
39
40              System.out.println("Hand 1:");
41              System.out.print(hand1);
42              System.out.println("Hand 2:");
43              System.out.print(hand2);
44
45              System.out.print("Who wins? (1 for Hand 1,
   2 for Hand 2, 0 for Tie): ");
46              int playerGuess = scanner.nextInt();
47
48              int comparison = hand1.compareTo(hand2);
49              int actualWinner;
50              String result;
51
52              if (comparison > 0) {
53                  actualWinner = 1;
54                  result = "Hand 1 wins!";
55              } else if (comparison < 0) {
56                  actualWinner = 2;
57                  result = "Hand 2 wins!";
58              } else {
59                  actualWinner = 0;
60                  result = "It's a tie!";
61              }
62
63              if (playerGuess == actualWinner) {
64                  System.out.println("Correct! " + result
   );
65                  correctGuesses++;
66              } else {
67                  System.out.println("Incorrect. " +
   result);
68              }
69
70              System.out.println("Cards remaining: " +
   deck.size());
```

```
71              System.out.println();
72
73              roundNumber++;
74          }
75
76          System.out.println("Game Over! Not enough
    cards left to play another round.");
77          System.out.println("You played " + (
    roundNumber - 1) + " rounds and won " + correctGuesses
     + " times");
78
79          scanner.close();
80      }
81  }
82
83
```

```java
package proj3;

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

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

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

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

```
163             for (int i = 0; i < dashCount; i++) {
164                 System.out.print("-");
165             }
166             System.out.println();
167             System.out.println();
168         }
169     }
170 
171     /**
172      * Initializes the test suite. Should be called before running any
173      * tests, so that passes and fails are correctly tallied.
174      */
175     public static void startTests()
176     {
177         System.out.println("Starting Tests");
178         System.out.println();
179         numTests = 0;
180         numFails = 0;
181     }
182 
183     /**
184      * Prints out summary data at end of tests. Should be called
185      * after all the tests have run.
186      */
187     public static void finishTests()
188     {
189         System.out.println("===============");
190         System.out.println("Tests Complete");
191         System.out.println("===============");
192         int numPasses = numTests - numFails;
193 
194         System.out.print(numPasses + "/" + numTests + " PASS ");
195         System.out.printf("(pass rate: %.1f%s)\n",
196                           100 * ((double) numPasses
```

```
196 ) / numTests,
197                             "%");
198
199         System.out.print(numFails + "/" + numTests +
    " FAIL ");
200         System.out.printf("(fail rate: %.1f%s)\n",
201                             100 * ((double) numFails) /
    numTests,
202                             "%");
203     }
204
205 }
206
```

```java
package proj3; // do not erase. Gradescope expects this.

import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Collections;

/**
 * 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)
    );

    /**
     * Constructs a PokerHand with the given cards.
     * @param cardList the cards that should be in this poker hand
     */
    public PokerHand(ArrayList<Card> cardList) {
        this.cards = new ArrayList<>(cardList);
    }

    /**
     * Add a card to the hand. Does nothing if the hand
```

```
33  already has 5 cards.
34       * @param card the card to add
35       */
36      public void addCard(Card card) {
37          if (cards.size() < 5) {
38              cards.add(card);
39          }
40      }
41
42      /**
43       * Get the i-th card from the hand.
44       * @param i the index of the card to get
45       * @return the card at index i, or null if i is
    invalid
46       */
47      public Card getIthCard(int i) {
48          if (i >= 0 && i < cards.size()) {
49              return cards.get(i);
50          }
51          return null;
52      }
53
54
55      /**
56       * Check if this hand is a flush (all cards same
    suit).
57       * @return true if the hand is a flush, else false
58       */
59      private boolean isFlush() {
60          HashSet<String> suits = new HashSet<>();
61          for (Card card : cards) {
62              suits.add(card.getSuit());
63          }
64          return suits.size() == 1;
65      }
66
67      /**
68       * Check if this hand is a straight (consecutive
```

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

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

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

```
167             return "flush";
168         } else if (flush) {
169             return "flush";
170         } else if (isFourOfAKind() || isFullHouse
    () || hasPairs(2)) {
171             return "two pair";
172         } else if (isThreeOfAKind() || hasPairs(1)) {
173             return "pair";
174         } else {
175             return "high card";
176         }
177     }
178 
179     /**
180      * Determines how this hand compares to another
    hand, returns
181      * positive, negative, or zero depending on the
    comparison.
182      *
183      * @param other The hand to compare this hand to
184      * @return a negative number if this is worth LESS
     than other, zero
185      * if they are worth the SAME, and a positive
    number if this is worth
186      * MORE than other
187      */
188     public int compareTo(PokerHand other) {
189         String myCategory = this.evaluate();
190         int myValue = CATEGORY_VALUE.get(myCategory);
191 
192         String otherCategory = other.evaluate();
193         int otherValue = CATEGORY_VALUE.get(
    otherCategory);
194 
195         int categoryDiff = myValue - otherValue;
196         if (categoryDiff != 0) {
197             return categoryDiff;
198         }
```

```
199
200         Map<Integer, Integer> myCounts = this.
    rankCounts();
201         Map<Integer, Integer> otherCounts = other.
    rankCounts();
202
203         List<int[]> myList = new ArrayList<>();
204         for (Map.Entry<Integer, Integer> entry :
    myCounts.entrySet()) {
205             myList.add(new int[]{entry.getValue(),
    entry.getKey()});
206         }
207
208         List<int[]> otherList = new ArrayList<>();
209         for (Map.Entry<Integer, Integer> entry :
    otherCounts.entrySet()) {
210             otherList.add(new int[]{entry.getValue(),
    entry.getKey()});
211         }
212
213         myList.sort((a, b) -> {
214             if (a[0] != b[0]) {
215                 return b[0] - a[0];
216             }
217             return b[1] - a[1];
218         });
219
220         otherList.sort((a, b) -> {
221             if (a[0] != b[0]) {
222                 return b[0] - a[0];
223             }
224             return b[1] - a[1];
225         });
226
227         for (int i = 0; i < myList.size(); 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             return 0;
237         }
238 
239 
240 }
241 
```

```java
package proj3;

/**
 * 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
69      public static void main(String[] args) {
70          Testing.startTests();
71          testGetRank();
72          testGetSuit();
73          testToString();
74          Testing.finishTests();
75      }
76  }
77
78
```

```java
package proj3;

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

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

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

```
131         for (int i = 0; i < 10; i++) {
132             deck.deal();
133         }
134         String shorterStr = deck.toString();
135         Testing.assertTrue("toString is shorter after
    dealing cards", shorterStr.length() < deckStr.length
    ());
136     }
137 
138 
139     public static void main(String[] args) {
140         Testing.startTests();
141         testDeckConstructor();
142         testSize();
143         testIsEmpty();
144         testDeal();
145         testGather();
146         testShuffle();
147         testToString();
148         Testing.finishTests();
149     }
150 }
151 
152 
```

```java
package proj3;

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 
261 
```