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

public class Card {


    private int cardRank;// Rank of the card
    private String cardSuit; //Suit of the card

    /**
     * Card Object Constructor
     * @param rank rank of the card
     * @param suit suit of the card
     */
    public Card(int rank, String suit){

        cardRank = rank;
        cardSuit = suit;

    }


    /**
     * getter for the rank of the card
     * @return int: rank
     */
    public int getCardRank(){
        return cardRank;
    }

    /**
     * Getter for the suit of the card
     *
     * @return String: suit
     */
    public String getCardSuit(){
        return cardSuit;
    }

    /**
     * format the Card as a string for printing and such
     *
     * @return String of the card
     */
    public @Override String toString(){

        String rankHolder = Integer.toString(cardRank);
        if(cardRank == 11){
            rankHolder = "Jack";
        }
        if(cardRank == 12){
            rankHolder = "Queen";
        }
        if(cardRank == 13){
            rankHolder = "King";
        }
        if(cardRank == 14){
            rankHolder = "Ace";
        }

        String cardString = "[ " + cardRank + " of " + cardSuit + " ]";
        return cardString;

    }

}

```

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

import java.util.*;
import java.util.concurrent.ThreadLocalRandom;

/**
 * Class for modeling a Deck of Cards
 */
public class Deck {

    private int DECK_SIZE =52;//standard deck size
    private ArrayList<Card> cardList = new ArrayList<Card
>(DECK_SIZE); //Arraylist of cards
    private int nextToDeal = 0;//intialized variable of
index we are drawing next card from

    /**
     * Constructor for a Deck object
     */
    public Deck() {
        generateDeck();//generates the deck
    }

    /**
     * Generates a standard 52 card deck
     * ranks: 2 to 14
     * suits: Hearts, Diamonds, Spades, Clubs
     */
    private void generateDeck(){
        for(int i=0; i<4; i++){
            for(int j=2; j<=14; j++) {
                Card card = null;
                if(i == 0){
                    card = new Card(j, "Hearts");
                }
                if(i == 1){
                    card = new Card(j,"Diamonds");
                }
                if(i == 2){
                    card = new Card(j, "Clubs");
                }
                if(i == 3){
                    card = new Card(j, "Spades");
                }
                cardList.add(card);
            }
        }
    }


    /**
     * shuffles the deck of cards
     */
    public void shuffle(){

        for(int i = nextToDeal; i < cardList.size(); i++){

            Card currentCard = cardList.get(i);

            int random = ThreadLocalRandom.current().nextInt(i, cardList.size());

            Card swapCard = cardList.get(random);

            cardList.set(i,swapCard);
            cardList.set(random, currentCard);
        }
    }


    /**
     * deals the next card in the deck
     * effiency: O(1)
     * @return previously undelt Card
     */
    public Card deal(){
        if(nextToDeal == this.size()){
            return null;
        }
        else {
            Card topCard = cardList.get(nextToDeal);
            nextToDeal++;
            return topCard;
        }

    }

    /**
     * return size of the deck; number of undelt cards
     * @return int value of number of cards
     */
    public int size(){
        int deckSize;
        deckSize = cardList.size() - nextToDeal;
        return deckSize;
    }

    /**
     * reset the next card to deal to the first in the deck
     */
    public void gather(){
        nextToDeal = 0;
    }

    /**
     * format the deck as a string for printing and such
     * @return String
     */
    public @Override String toString(){
        String str = "";
        for(int i = nextToDeal; i <cardList.size(); i++){
            str += cardList.get(i).toString();
            str += "\n";
        }
        return str;
    }

}

```

```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;
    }

    /**
     * Each of the assertEquals methods tests whether the actual
     * result equals the expected result. If it does, then the test
     * passes, otherwise it fails.
     *
     * The only difference between these methods is the types of the
     * parameters.
     *
     * All take a String message and two values of some other type to
     * compare:
     *
     * @param message
     *            a message or description of the test
     * @param expected
     *            the correct, or expected, value
     * @param actual
     *            the actual value
     */
    public static void assertEquals(String message, boolean expected,
                                    boolean actual)
    {
        printTestCaseInfo(message, "" + expected, "" + actual);
        if (expected == actual) {
            pass();
        } else {
            fail(message);
        }
    }

    public static void assertEquals(String message, int expected, int actual)
    {
        printTestCaseInfo(message, "" + expected, "" + actual);
        if (expected == actual) {
            pass();
        } else {
            fail(message);
        }
    }

    public static void assertEquals(String message, Object expected,
                                    Object actual)
    {
        String expectedString = "<<null>>";
        String actualString = "<<null>>";
        if (expected != null) {
            expectedString = expected.toString();
        }
        if (actual != null) {
            actualString = actual.toString();
        }
        printTestCaseInfo(message, expectedString, actualString);

        if (expected == null) {
            if (actual == null) {
                pass();
            } else {
                fail(message);
            }
        } else if (expected.equals(actual)) {
            pass();
        } else {
            fail(message);
        }
    }

    /**
     * Asserts that a given boolean must be true.  The test fails if
     * the boolean is not true.
     *
     * @param message The test message
     * @param actual The boolean value asserted to be true.
     */
    public static void assertTrue(String message, boolean actual)
    {
        assertEquals(message, true, actual);
    }

    /**
     * Asserts that a given boolean must be false. The test fails if
     * the boolean is not false (i.e. if it is true).
     *
     * @param message The test message
     * @param actual The boolean value asserted to be false.
     */
    public static void assertFalse(String message, boolean actual)
    {
        assertEquals(message, false, actual);
    }

    private static void printTestCaseInfo(String message, String expected,
                                          String actual)
    {
        if (VERBOSE) {
            System.out.println(message + ":");
            System.out.println("expected: " + expected);
            System.out.println("actual:   " + actual);
        }
    }

    private static void pass()
    {
        numTests++;

        if (VERBOSE) {
            System.out.println("--PASS--");
            System.out.println();
        }
    }

    private static void fail(String description)
    {
        numTests++;
        numFails++;

        if (!VERBOSE) {
            System.out.print(description + "   ");
        }
        System.out.println("--FAIL--");
        System.out.println();
    }

    /**
     * Prints a header for a section of tests.
     *
     * @param sectionTitle The header that should be printed.
     */
    public static void testSection(String sectionTitle)
    {
        if (VERBOSE) {
            int dashCount = sectionTitle.length();
            System.out.println(sectionTitle);
            for (int i = 0; i < dashCount; i++) {
                System.out.print("-");
            }
            System.out.println();
            System.out.println();
        }
    }

    /**
     * Initializes the test suite. Should be called before running any
     * tests, so that passes and fails are correctly tallied.
s      */
    public static void startTests()
    {
        System.out.println("Starting Tests");
        System.out.println();
        numTests = 0;
        numFails = 0;
    }

    /**
     * Prints out summary data at end of tests.  Should be called
     * after all the tests have run.
     */
    public static void finishTests()
    {
        System.out.println("==============");
        System.out.println("Tests Complete");
        System.out.println("==============");
        int numPasses = numTests - numFails;

        System.out.print(numPasses + "/" + numTests + " PASS ");
        System.out.printf("(pass rate: %.1f%s)\n",
                          100 * ((double) numPasses) / numTests,
                          "%");

        System.out.print(numFails + "/" + numTests + " FAIL ");
        System.out.printf("(fail rate: %.1f%s)\n",
                          100 * ((double) numFails) / numTests,
                          "%");
    }

}

```

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

import java.util.*;

public class PokerHand {

    private int MAX_HAND_SIZE = 5;
    private ArrayList<Card> cardsInHand; //all the cards in the hand

    private ArrayList<Integer> allRanks = new ArrayList<Integer>(); //for seperating out the ranks of the cards
    private ArrayList<String> allSuits = new ArrayList<String>(); //for seperating out the suits

    private ArrayList<Integer> pairRanks = new ArrayList<Integer>(); //for seperating out the pairs
    private ArrayList<Integer> highcardRanks = new ArrayList<Integer>(); //for seperating out the non-pairs

    private TreeMap<Integer, Integer> rankOccurances = new TreeMap<Integer, Integer>(Collections.reverseOrder());// experimenting with a new data structure, makes sorting fuctions simplier

    /**
     * A Constructer for a PokerHand Object
     *
     * @param cardList cards that will make up the PokerHand
     */
    public PokerHand(ArrayList<Card> cardList) {
        cardsInHand = cardList;
    }

    /**
     * add a card to the Poker Hand if there are less than 5 cards in the hand
     * otherwise do nothing
     *
     * @param card card being added to the PokerHand
     */
    public void addCard(Card card) {

        if (cardsInHand.size() < MAX_HAND_SIZE) {
            cardsInHand.add(card);
        }
    }

    /**
     * return the card in the pokerHand at the given index
     *
     * @param index index of card being retrieved
     * @return Card
     */
    public Card get_ith_card(int index) {
        if (index >= 0 && index < cardsInHand.size() - 1) {
            return cardsInHand.get(index);
        } else {
            return null;
        }
    }

    /**
     * override the toString function to turn a PokerHand into a properly formatted string
     * @return String
     */
    public @Override String toString() {
        String str = "";
        for (int i = 0; i < cardsInHand.size(); i++) {
            str += cardsInHand.get(i).toString();
            str += "\n";
        }
        return str;
    }

    /**
     * Determines how this hand compares to another hand, returns
     * positive, negative, or zero depending on the comparison.
     *
     * @param other The hand to compare this hand to
     * @return a negative number if this is worth LESS than other, zero
     * if they are worth the SAME, and a positive number if this is worth
     * MORE than other
     */
    public int compareTo(PokerHand other) {

        //organize the hands and determine their types
        //this hand
        this.getHandData();
        Integer hand1Type = this.handType();
        //otherhand
        other.getHandData();
        Integer hand2Type = other.handType();

        int handTypeComparison = hand1Type.compareTo(hand2Type);

        if(handTypeComparison != 0){
            return handTypeComparison;
         }

        else {  //if hands are of the same type...
            if(!pairRanks.isEmpty()){ //if there are pairs to compare...
                int pairCompare = this.tieBreaker(this.pairRanks, other.pairRanks); //compare them
                if(pairCompare == 0){ //if the pair values are equal
                    return this.tieBreaker(this.highcardRanks, other.highcardRanks); // return the highcard comparison
                }
                else{ //otherwise return the pair comparison
                    return pairCompare;
                }

            }
            else{ //if there are no pairs to compare, just return the highcard comparison
                return this.tieBreaker(this.highcardRanks, other.highcardRanks);
            }
        }
    }

    /**
     * Determing the type of the hand. Flush, 2pair, 1pair, or highcard
     *
     * @return Integer : 4 if flush, 3 if 2pair, 2 if 1pair, 1 if highcard
     */
    private Integer handType() {

        boolean isFlush = flushCheck();
        if(isFlush) {
            return 4; //FLUSH
        }
        if (pairRanks.size() == 2) { // if you have 2 pairs
            return 3; //2Pair
        }
        if (pairRanks.size() == 1) { //if you have 1 pair
            return 2; //1 PAIR
        }
        else { //If its not a flush, 2pair, or 1pair it has to be....
            return 1; //HIGHCARD
        }

    }

    /**
     * Breaks up all the hand data into managable chunks:
     * seperates suits and ranks into their own ArrayLists,
     * creates a Treemap of the ranks and their # of occurances,
     * uses the Treemap to fill ArrayLists with pair values, and non-pair(highcard) values
     *
     */
    private void getHandData(){

        int size = cardsInHand.size(); //size of the hand

        for (int i = 0; i < size; i++) {
            Card currentCard = cardsInHand.get(i);// current card we are pulling data from
            allSuits.add(currentCard.getCardSuit()); // adding the current cards suit to the suit array
            allRanks.add(currentCard.getCardRank()); // adding the current cards rank to the rank array
        }

        //sort allRanks into a TreeMap with Key = Rank & Value = instances of the rank
        for (Integer i : allRanks) {
            Integer j = rankOccurances.get(i);
            rankOccurances.put(i, (j == null) ? 1 : j + 1);
        }

        //sort rankOccurances by pairs and non-pairs(highcards)
        for(Integer key : rankOccurances.keySet()){
            if(rankOccurances.get(key) == 4){ //2pair
                pairRanks.add(key);
                pairRanks.add(key);
            }
            if(rankOccurances.get(key) == 2 || rankOccurances.get(key) == 3){
                pairRanks.add(key);
            }
            else{
                highcardRanks.add(key);
            }
        }
    }

    /**
     * compares two ArrayLists of ranks and determines which has the first instance of a greater value
     *
     * @param theseRanks Arraylist of ranks from this hand
     * @param otherRanks Arraylist of ranks from other hand
     * @return int 1 if theseRanks is greater, -1 if otherRanks is greater, 0 if all ranks are the same
     */
    private int tieBreaker(ArrayList<Integer> theseRanks, ArrayList<Integer> otherRanks) {

        //compare each rank
        for (int i = 0; i < theseRanks.size(); i++) {
            int currentCompare = theseRanks.get(i).compareTo(otherRanks.get(i)); //compare current index
            if (currentCompare != 0) { //if the current index ranks are different...
                return currentCompare; //return the comparison
            }
        }
        return 0; //you make it through all ranks and they are all the same
    }

    /**
     * checks if the hand is a flush
     *
     * @return true if hand is a flush, false if not
     */
    private boolean flushCheck(){

        //checking for a flush
        String checkSuit = allSuits.get(1);//a suit present in the hand
        if(Collections.frequency(allSuits, checkSuit) == cardsInHand.size()){
            return true;
        }
        else{
            return false;
        }
    }
}
```

```java
package proj3;
import org.junit.Test;

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

/**
 * Author: Ian Sulley
 *
 * Honor Code: 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
 */

/**
 * Testing Class for PokerHand compareTo()
 */
public class PokerComparisonTests {

    public static void main(String[] args) {

        test_all();
    }


    //#####TESTS###########
    public static void test_all(){
        Testing.startTests();
        test_all_flushes();
        test_all_two_pair();
        test_all_pair();
        Testing.finishTests();
    }


//#####FLUSH TESTS#####

    public static void test_all_flushes() {
        Testing.startTests();
        compare_flushes1();
        compare_flushes2();
        compare_flushes_tie();
        compare_flush_2pair();
        compare_flush_pair();
        compare_flush_hi();
        Testing.finishTests();
    }

    //# Flush1 vs Flush2 (Flush 1 wins highcard is greater
 )
     public static void compare_flushes1() {
         ArrayList<Card> hand1array = new ArrayList<Card>(Arrays.asList(new Card(13, "S"), new Card(12, "S"), new Card(9, "S"), new Card(7, "S"), new Card(3, "S")));
         PokerHand hand1 = new PokerHand(hand1array);
         PokerHand hand2 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(4, "C"), new Card(2, "C"), new Card(7, "C"), new Card(5, "C"), new Card(10, "C"))));

         int expected_answer = 1;
         int actual_answer = hand1.compareTo(hand2);
         Testing.assertEquals("Testing Flush1 vs Flush2 (Flush 1 wins; Highcard is greater)",
                 expected_answer,
                 actual_answer);
     }

     //# Flush1 vs Flush2 (Flush 2 wins highcard is greater)
     public static void compare_flushes2() {
         PokerHand hand1 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(4, "C"), new Card(2, "C"), new Card(7, "C"), new Card(5, "C"), new Card(10, "C"))));
         PokerHand hand2 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(13, "S"), new Card(12, "S"), new Card(9, "S"), new Card(7, "S"), new Card(3, "S"))));

         int expected_answer = -1;
         int actual_answer = hand1.compareTo(hand2);
         Testing.assertEquals("Testing Flush1 vs Flush2 (Flush 2 wins highcard is greater)",
                 expected_answer,
                 actual_answer);
     }

     //# Flush1 vs Flush2 (Tie)
     public static void compare_flushes_tie() {
         PokerHand hand1 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(4, "C"), new Card(2, "C"), new Card(7, "C"), new Card(5, "C"), new Card(10, "C"))));
         PokerHand hand2 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(7, "S"), new Card(5, "S"), new Card(4, "S"), new Card(2, "S"), new Card(10, "S"))));

         int expected_answer = 0;
         int actual_answer = hand1.compareTo(hand2);
         Testing.assertEquals("Testing Flush1 vs Flush2 Tie",
                 expected_answer,
                 actual_answer);
     }

     //# Flush vs 2 pair
     public static void compare_flush_2pair() {
         PokerHand hand1 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(7, "S"), new Card(5, "S"), new Card(11, "S"), new Card(2, "S"), new Card(10, "S"))));
         PokerHand hand2 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"), new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));

         int expected_answer = 1;
         int actual_answer = hand1.compareTo(hand2);
         Testing.assertEquals("Testing Flush1 vs 2pair (Flush 1 wins)",
                 expected_answer,
                 actual_answer);
     }

     //# Flush vs pair
     public static void compare_flush_pair() {
         PokerHand hand1 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(7, "S"), new Card(5, "S"), new Card(4, "S"), new Card(2, "S"), new Card(10, "S"))));
         PokerHand hand2 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"), new Card(9, "S"), new Card(10, "C"), new Card(6, "D"))));

         int expected_answer = 1;
         int actual_answer = hand1.compareTo(hand2);
         Testing.assertEquals("Testing Flush1 vs pair (Flush 1 wins)",
                 expected_answer,
                 actual_answer);
     }

     //# Flush vs high Card
     public static void compare_flush_hi() {
         PokerHand hand1 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(7, "S"), new Card(5, "S"), new Card(4, "S"), new Card(2, "S"), new Card(10, "S"))));
         PokerHand hand2 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(3, "H"), new Card(4, "D"), new Card(10, "S"), new Card(8, "C"), new Card(6, "D"))));

         int expected_answer = 1;
         int actual_answer = hand1.compareTo(hand2);
         Testing.assertEquals("Testing Flush1 vs highcard (Flush 1 wins)",
                 expected_answer,
                 actual_answer);
     }

 //#####2 PAIR TESTS#####

     public static void test_all_two_pair() {
         Testing.startTests();
         compare_2pair_flush();
         compare_2pair_2pair_1();
         compare_2pair_2pair_2();
         compare_2pair_2pair_3();
         compare_2pair_2pair_4();
         Testing.finishTests();
     }

     //# 2pair vs Flush
     public static void compare_2pair_flush() {
         PokerHand hand1 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"), new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));
         PokerHand hand2 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(7, "S"), new Card(5, "S"), new Card(11, "S"), new Card(2, "S"), new Card(10, "S"))));

         int expected_answer = -1;
         int actual_answer = hand1.compareTo(hand2);
         Testing.assertEquals("Testing 2pair vs Flush",
                 expected_answer,
                 actual_answer);
     }

     //# 2pair1 vs 2pair2 (2pair1 wins higher of pair values is greater)
     public static void compare_2pair_2pair_1() {
         PokerHand hand1 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(4, "H"), new Card(6, "D"), new Card(10, "S"), new Card(10, "C"), new Card(4, "D"))));
         PokerHand hand2 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"), new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));

         int expected_answer = 1;
         int actual_answer = hand1.compareTo(hand2);
         Testing.assertEquals("Testing 2pair1 vs 2pair2 (2pair1 wins higher of pair values is greater)",
                 expected_answer,
                 actual_answer);
     }

     //# 2pair1 vs 2pair2 (2pair2 wins higher of pair values is greater)
     public static void compare_2pair_2pair_2() {
         PokerHand hand1 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"), new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));
         PokerHand hand2 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(14, "H"), new Card(14, "D"), new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));

         int expected_answer = -1;
         int actual_answer = hand1.compareTo(hand2);
         Testing.assertEquals("Testing 2pair1 vs 2pair2 (2pair2 wins higher of pair values is greater)",
                 expected_answer,
                 actual_answer);
     }

     //# 2pair1 vs 2pair2 (2pair1 wins lower of pair values is greater)
     public static void compare_2pair_2pair_3() {
         PokerHand hand1 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"), new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));
         PokerHand hand2 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(3, "H"), new Card(4, "D"), new Card(8, "S"), new Card(8, "C"), new Card(3, "D"))));

         int expected_answer = 1;
         int actual_answer = hand1.compareTo(hand2);
         Testing.assertEquals("2pair1 vs 2pair2 (2pair1 wins lower of pair values is greater)",
                 expected_answer,
                 actual_answer);
     }

     //# 2pair1 vs 2pair2 (2pair2 wins lower of pair values is greater)
     public static void compare_2pair_2pair_4() {
         PokerHand hand1 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(3, "H"), new Card(4, "D"), new Card(8, "S"), new Card(8, "C"), new Card(3, "D"))));
         PokerHand hand2 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(4, "H"), new Card(4, "D"), new Card(8, "S"), new Card(8, "C"), new Card(6, "D"))));

         int expected_answer = -1;
         int actual_answer = hand1.compareTo(hand2);
         Testing.assertEquals("Testing 2pair1 vs 2pair2 (2pair2 wins lower of pair values is greater)",
                 expected_answer,
                 actual_answer);
     }
```

```

//#####PAIR TESTS#####

    public static void test_all_pair() {
        Testing.startTests();
        compare_pair_pair_1();
        compare_pair_pair_2();
        compare_pair_pair_3();
        compare_pair_pair_4();
        Testing.finishTests();
    }


    //# pair1 vs pair2 (pair1 wins; high pair)
    public static void compare_pair_pair_1() {
        PokerHand hand1 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"),
new Card(9, "S"), new Card(10, "C"), new Card(6, "D"))));
        PokerHand hand2 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(2, "H"), new Card(4, "D"),
new Card(9, "S"), new Card(9, "C"), new Card(6, "D"))));

        int expected_answer = 1;
        int actual_answer = hand1.compareTo(hand2);
        Testing.assertEquals("pair1 vs pair2 (pair1 wins
; high pair)",
                expected_answer,
                actual_answer);
    }

    //# pair1 vs pair2 (pair2 wins; high pair)
    public static void compare_pair_pair_2() {
        PokerHand hand1 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"),
new Card(9, "S"), new Card(10, "C"), new Card(6, "D"))));
        PokerHand hand2 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(12, "H"), new Card(4, "D"),
new Card(9, "S"), new Card(12, "C"), new Card(6, "D"))));

        int expected_answer = -1;
        int actual_answer = hand1.compareTo(hand2);
        Testing.assertEquals("Testing pair1 vs pair2 (
pair2 wins; high pair)",
                expected_answer,
                actual_answer);
    }

    //# pair1 vs pair2 (pair1 wins; highcard)
    public static void compare_pair_pair_3() {
        PokerHand hand1 = new PokerHand(new ArrayList<
Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"),
new Card(12, "S"), new Card(10, "C"), new Card(6, "D"
```

```java
))));
        PokerHand hand2 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"), new Card(9, "S"), new Card(10, "C"), new Card(6, "D"))));

        int expected_answer = 1;
        int actual_answer = hand1.compareTo(hand2);
        Testing.assertEquals("Testing pair1 vs pair2 (pair1 wins; highcard)",
                expected_answer,
                actual_answer);
    }

    //# pair1 vs pair2 (pair2 wins; highcard)
    public static void compare_pair_pair_4() {
        PokerHand hand1 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(10, "H"), new Card(4, "D"), new Card(9, "S"), new Card(10, "C"), new Card(6, "D"))));
        PokerHand hand2 = new PokerHand(new ArrayList<Card>(Arrays.asList(new Card(10, "H"), new Card(13, "D"), new Card(9, "S"), new Card(10, "C"), new Card(6, "D"))));

        int expected_answer = -1;
        int actual_answer = hand1.compareTo(hand2);
        Testing.assertEquals("Testing pair1 vs pair2 (pair2 wins; highcard",
                expected_answer,
                actual_answer);
    }
}





/*


 _______  _     _  ______     ______  _     _  _____
(__ _ __)(_)   (_)(______)   (______)(_)   (_)(_____)
   (_)   (_)___(_)(_)__      (_)__   (__)_ (_)(_)  (_)
   (_)   (_______)(____)     (____)  (_)(_)(_)(_)  (_)
   (_)   (_)   (_)(_)____    (_)____ (_)  (__)(_)__(_)
   (_)   (_)   (_)(______)   (______)(_)   (_)(_____)


 */




```