```
Beginning tests for yey11 at Thu Oct 11 19:18:19 AEDT 2018

==========
SelectHand.java not submitted
Not testing SelectHand

==========
compiling HandValue.java
Testing HandValue:
                    01) java HandValue ...cards...  PASS  100
                    02) java HandValue ...cards...  PASS  100
                    03) java HandValue ...cards...  PASS  100
                    04) java HandValue ...cards...  PASS  100
                    05) java HandValue ...cards...  PASS  100
                    06) java HandValue ...cards...  PASS  100
                    07) java HandValue ...cards...  PASS  100
                    08) java HandValue ...cards...  PASS  100
                    09) java HandValue ...cards...  PASS  100
                    10) java HandValue ...cards...  PASS  100
                    11) java HandValue ...cards...  PASS  100
                    12) java HandValue ...cards...  PASS  100
                    13) java HandValue ...cards...  PASS  100
                    14) java HandValue ...cards...  PASS  100
                    15) java HandValue ...cards...  PASS  100
                    16) java HandValue ...cards...  PASS  100
                    17) java HandValue ...cards...  PASS  100
                    18) java HandValue ...cards...  PASS  100
                    19) java HandValue ...cards...  PASS  100
                    20) java HandValue ...cards...  PASS  100
                    21) java HandValue ...cards...  PASS  100
                    22) java HandValue ...cards...  PASS  100
                    23) java HandValue ...cards...  PASS  100
                    24) java HandValue ...cards...  PASS  100
                    25) java HandValue ...cards...  PASS  100
                    26) java HandValue ...cards...  PASS  100
                    27) java HandValue ...cards...  PASS  100
                    28) java HandValue ...cards...  PASS  100
                    29) java HandValue ...cards...  PASS  100
                    30) java HandValue ...cards...  PASS  100
                    31) java HandValue ...cards...  PASS  100
                    32) java HandValue ...cards...  PASS  100
                    33) java HandValue ...cards...  PASS  100
                    34) java HandValue ...cards...  PASS  100
                    35) java HandValue ...cards...  PASS  100
                    36) java HandValue ...cards...  PASS  100
                    37) java HandValue ...cards...  PASS  100
                    38) java HandValue ...cards...  PASS  100
                    39) java HandValue ...cards...  PASS  100
                    40) java HandValue ...cards...  PASS  100
                    41) java HandValue ...cards...  PASS  100
                    42) java HandValue ...cards...  PASS  100
                    43) java HandValue ...cards...  PASS  100
                    44) java HandValue ...cards...  PASS  100
                    45) java HandValue ...cards...  PASS  100
                    46) java HandValue ...cards...  PASS  100
                    47) java HandValue ...cards...  PASS  100
                    48) java HandValue ...cards...  PASS  100
                    49) java HandValue ...cards...  PASS  100
                    50) java HandValue ...cards...  PASS  100
```

```
Total HandValue score  : 5000 / 5000

Final score for project correctness:  100%

Testing completed Thu Oct 11 19:18:47 AEDT 2018
```

```
/** This class provides a single static method to compute all
 * combinations of objects in the input.  It also provides a main
 * method for experimentation.
 *
 * @author Peter Schachte <schachte@unimelb.edu.au>
 */
import java.lang.reflect.Array;

public class Combinations {

    /** This method computes all the combinations of objects from the
     * input array.  If the input array has n elements, the output array
     * has 2^n arrays, each containing from 0 to n elements taken from the
     * input array.  Elements in the output arrays are included in the order
     * they appear in the input array.  Specifically, element i of the
     * output array contains element j of the input array if i / 2^j is
     * odd, where / is integer division rounding down, so element 0 of the
     * result is empty and element 2^n - 1 of the result has all the
     * elements of the input.
     *
     * @param <T> the type of the array elements
     * @param list an array of the elements to compute the combinations of
     * @return an array of the "subarrays" of the input
     *
     * @see <a href="http://en.wikipedia.org/wiki/Combination">
     *      the wikipedia page for combinations</a>
     */
    @SuppressWarnings({"unchecked"})
    public static <T> T[][] combinations(T[] list) {
        T[][] combos = (T[][])Array.newInstance(list.getClass(),
            (int) Math.pow(2, list.length));

        for (int i = 0 ; i < combos.length ; ++i) {
            int count = 0;
            for (int j = 0 ; j < list.length ; ++j) {
                if ((i & 1<<j) != 0) ++count;
            }
            combos[i] = (T[])Array.newInstance(list.getClass().getComponentType(
), count);
            count = 0;
            for (int j = 0 ; j < list.length ; ++j) {
                if ((i & 1<<j) != 0) {
                    combos[i][count] = list[j];
                    ++count;
                }
            }
        }
        return combos;
    }

    /** A simple main method to allow experimentation with this
     *  method.  It simply prints out all the combinations of the
     *  command line arguments, one per line.
     */
    public static void main(String[] args) {
        String[][] lines = Combinations.combinations(args);
        for (String[] line : lines) {
            for (String str : line) {
                System.out.print(str + " ");
```

```
            }
            System.out.println();
        }
    }
}
```

```
/** A playing card rank type designed for cribbage.  It supports a single-
 *  character abbreviation for each rank, as well as providing the face
 *  value of a rank (ACE=1, KING, QUEEN, and JACK=10, other ranks are their
 *  face value), needed for counting 15s in a cribbage hand.  Also provides
 *  methods to get the next smaller and larger rank of a given rank.
 *
 * @author Peter Schachte schachte@unimelb.edu.au
 */
public enum CribbageRank {
    ACE('A'),
    TWO('2'),
    THREE('3'),
    FOUR('4'),
    FIVE('5'),
    SIX('6'),
    SEVEN('7'),
    EIGHT('8'),
    NINE('9'),
    TEN('T'),
    JACK('J'),
    QUEEN('Q'),
    KING('K');

    /** Single character abbreviation used to briefly print the rank. */
    private final char abbrev;

    /** @return the single-character abbreviation for this rank. */
    public char abbrev() {
        return abbrev;
    }

    /** @return the face value of the rank for counting 15's in cribbage
     *  (ACE=1, KING, QUEEN, and JACK=10, other ranks are their face value).
     */
    public int faceValue() {
        return Math.min(this.ordinal()+1, 10);
    }

    /** @return the next higher rank */
    public CribbageRank nextHigher() {
        int value = this.ordinal() + 1;
        return value >= values().length ? null : values()[value];
    }

    /** @return the next lower rank */
    public CribbageRank nextLower() {
        int value = this.ordinal() - 1;
        return value < 0 ? null : values()[value];
    }

    /** Construct a rank.
     *  @param abbrev the single-character abbreviation for this rank
     */
    CribbageRank(char abbrev) {
        this.abbrev = abbrev;
    }

    /** @return The rank as a single-character string. */
    @Override
```

```
    public String toString() {
        return Character.toString(abbrev);
    }
}
```

```
/** This class provides a series of static methods to calculate the value of
 *  a hand for Cribbage Game. The methods fifteens, pairs, runs, flushes and
 *  oneForHisNob represents the five rules to calculate the score of a hand.
 *  Other methods are used for converting the input from the command line into
 *  appropriate types for the methods of the rules.
 */
public class HandValue {

    /** Integer used to store the value of a hand for the game. */
    private static int score = 0;

    /** This method calculates and prints the total score of a hand for
     *  the game as an output. It receives 5 cards on the command line,
     *  while the first 4 cards for the hand and the 5th card for the start
     *  card. Cards should be entered on the command line as two-character
     *  strings, the first being an upper-case A for Ace, K for King, Q for
     *  Queen, J for Jack, T for Ten, or digit between 2 and 9 for ranks 2-9.
     *  The second character should be a C for Clubs (âM-^Y£), D for Diamonds (â
M-^Y¦),
     *  H for Hearts (âM-^Y¥), or a S for Spades (âM-^Y ).
     *
     *  @param args an array of the double-character strings on the command line
.
     */
    public static void main(String[] args) {

        /** The suit is used to store the card suits. */
        char[] suits = getSuit(args);

        /** The ranks is used to store the card ranks. */
        CribbageRank[] ranks = getRank(args);

        /** The comb is used to store the combinations of card ranks. */
        CribbageRank[][] comb = Combinations.combinations(ranks);

        /** The sortComb is used to store the combinations of sorted card ranks.
 */
        CribbageRank[][] sortComb = sortComb(comb);

        score = fifteens(comb) + pairs(sortComb) + runs(sortComb)
            + flushes(suits) + oneForHisNob(ranks, suits);
        System.out.println(score);
    }

    /** This method calculates the points based on the rule of fifteens.
     *  @param comb an array of the "sub-arrays" of the input card ranks.
     *  @return the integer points of fifteens.
     */
    private static int fifteens(CribbageRank[][] comb) {
        int points = 0;
        for (CribbageRank[] subComb : comb) {
            int sum = 0;
            for (CribbageRank rank : subComb) {
                sum += rank.faceValue();
            }
            if (sum == 15) {
                points += 2;
            }
        }
```

```
        return points;
    }

    /** This method calculates the points based on the rule of pairs.
     *  @param sortComb an array of the "sub-arrays" of the sorted card ranks.
     *  @return the integer points of pairs.
     */
    private static int pairs(CribbageRank[][] sortComb) {
        int points = 0;
        for (CribbageRank[] subComb : sortComb) {
            if (subComb.length == 2 && (subComb[0] == subComb[1])) {
                points += 2;
            }
        }
        return points;
    }

    /** This method calculates the points based on the rule of runs.
     *  @param sortComb an array of the "sub-arrays" of the sorted card ranks.
     *  @return the integer points of runs.
     */
    private static int runs(CribbageRank[][] sortComb) {
        int fiveRuns = 0;
        int fourRuns = 0;
        int threeRuns = 0;
        for (CribbageRank[] subComb : sortComb) {
            switch (subComb.length) {
                case 5:
                    if ((subComb[0].nextHigher() == subComb[1])
                        && (subComb[1].nextHigher() == subComb[2])
                        && (subComb[2].nextHigher() == subComb[3])
                        && (subComb[3].nextHigher() == subComb[4])) {
                        fiveRuns++;
                    }
                    break;
                case 4:
                    if ((subComb[0].nextHigher() == subComb[1])
                        && (subComb[1].nextHigher() == subComb[2])
                        && (subComb[2].nextHigher() == subComb[3])) {
                        fourRuns++;
                    }
                    break;
                case 3:
                    if ((subComb[0].nextHigher() == subComb[1])
                        && (subComb[1].nextHigher() == subComb[2])) {
                        threeRuns++;
                    }
                    break;
            }
        }
        return fiveRuns > 0? 5:(fourRuns > 0? 4 * fourRuns:(threeRuns > 0? 3 * t
hreeRuns:0));
    }

    /** This method calculates the points based on the rule of flushes.
     *  @param suits a char array of the input card suits.
     *  @return the integer points of flushes.
     */
    private static int flushes(char[] suits) {
```

```
        int points = 0;
        boolean sameHandSuits = true;
        for (int i = 1; i < suits.length - 1; i++) {
            if (suits[i - 1] != suits[i]) {
                sameHandSuits = false;
            }
        }
        if (sameHandSuits) {
            points = 4;
            if (suits[0] == suits[suits.length - 1]) {
                points++;
            }
        }
        return points;
    }

    /** This method calculates the points based on the rule of one for his nob.
     *  @param ranks an CribbageRank array of the input card ranks.
     *  @param suits a char array of the input card suits.
     *  @return the integer points of on for his nob.
     */
    private static int oneForHisNob(CribbageRank[] ranks, char[] suits) {
        int points = 0;
        for (int i = 0; i < ranks.length - 1; i++) {
            if (ranks[i].abbrev() == 'J' && suits[i] == suits[suits.length - 1])
 {
                points = 1;
            }
        }
        return points;
    }

    /** This method constructs an CribbageRank array of input card ranks.
     *  @param args a double-character string array of input card ranks and suit
s.
     *  @return an CribbageRank array of input card ranks.
     */
    private static CribbageRank[] getRank(String[] args) {
        CribbageRank[] ranks = new CribbageRank[args.length];
        for (int i = 0; i < ranks.length; i++) {
            switch (args[i].charAt(0)) {
                case 'A':
                    ranks[i] = CribbageRank.ACE;
                    break;
                case '2':
                    ranks[i] = CribbageRank.TWO;
                    break;
                case '3':
                    ranks[i] = CribbageRank.THREE;
                    break;
                case '4':
                    ranks[i] = CribbageRank.FOUR;
                    break;
                case '5':
                    ranks[i] = CribbageRank.FIVE;
                    break;
                case '6':
                    ranks[i] = CribbageRank.SIX;
                    break;
```

```
                case '7':
                    ranks[i] = CribbageRank.SEVEN;
                    break;
                case '8':
                    ranks[i] = CribbageRank.EIGHT;
                    break;
                case '9':
                    ranks[i] = CribbageRank.NINE;
                    break;
                case 'T':
                    ranks[i] = CribbageRank.TEN;
                    break;
                case 'J':
                    ranks[i] = CribbageRank.JACK;
                    break;
                case 'Q':
                    ranks[i] = CribbageRank.QUEEN;
                    break;
                case 'K':
                    ranks[i] = CribbageRank.KING;
                    break;
                default:
                    System.out.println("Input wrong!");
                    break;
            }
        }
        return ranks;
    }

    /** This method constructs a char array of input card suits.
     *  @param args a double-character string array of input card ranks and suit
s.
     *  @return an char array of input card suits.
     */
    private static char[] getSuit(String[] args) {
        char[] suits = new char[args.length];
        for (int i = 0; i < suits.length; i++) {
            suits[i] = args[i].charAt(1);
        }
        return suits;
    }

    /** This method sorts an array of input card ranks from low to high.
     *  @param comb an array of the "sub-arrays" of the input card ranks.
     *  @return an array of the "sub-arrays" of the sorted card ranks.
     */
    private static CribbageRank[][] sortComb(CribbageRank[][] comb) {
        for (int i = 0; i < comb.length; i++) {
            for (int j = 0; j < comb[i].length - 1; j++) {
                for (int k = 0; k < comb[i].length - 1 - j; k++) {
                    if (comb[i][k].ordinal() > comb[i][k + 1].ordinal()) {
                        CribbageRank temp = comb[i][k];
                        comb[i][k] = comb[i][k + 1];
                        comb[i][k + 1] = temp;
                    }
                }
            }
        }
        return comb;
```

```
    }
}
```