// Text in **BOLD** needs to be examined and modified.

// Oliver Kullmann, 26.11.2009 (Swansea)

/\* Copyright 2009 Oliver Kullmann \*/

#include <iostream>

class Card {

// A "card" as a pair of (card-)rank and suit.

public: int num\_cards = **Suit.num\_suites** \* **CardRank.num\_ranks**;

public: CardRank; rank;

public: Suit; suit;

public: Card(CardRank r, final Suit s) {

rank = r;

suit = s;

}

public: Card(int i) {

assert i >= 0;

assert i < num\_cards;

rank = new CardRank(i % **CardRank.num\_ranks**);

suit = new Suit(i / **CardRank.num\_ranks**);

}

public: int index() {

return **suit.index** \* **CardRank.num\_ranks** + **rank.index**;

}

public: boolean; equals(Card c) {

return **c.rank.equals(rank) && c.suit.equals(suit)**;

}

// Demonstration of functionality:

int main() {

using std::cout;

cout << new Card(0);

cout << new Card(51);

Card c1 = new Card(new CardRank("Jack"), new Suit("Spades"));

cout << c1;

// Demonstration that here references are harmless, since CardRank, Suit

// and also Card behave similar to value-types (they are non-mutable):

CardRank cr2 = new CardRank("5");

Suit s2 = new Suit("Hearts");

Card c2 = new Card(cr2, s2);

cout << c2;

cr2 = new CardRank("King");

s2 = new Suit("Clubs");

cout << c2; // still the same, since the old cr2- and s2-objects didn't change

cout << new Card(cr2,s2);

return 0;

}

}

// Oliver Kullmann, 26.11.2009 (Swansea)

/\* Copyright 2009 Oliver Kullmann \*/

#include <iostream>

class CardRank {

// Wrapper class around card ranks as integer indices.

// Note that a smaller index means a higher card.

public: int num\_ranks = 13;

public: int ace = 0;

public: int king = 1;

public: int queen = 2;

public: int jack = 3;

public: int ten = 4;

public: int nine = 5;

public: int eight = 6;

public: int seven = 7;

public: int six = 8;

public: int five = 9;

public: int four = 10;

public: int three = 11;

public: int two = 12;

public: int index;

public CardRank(int i) {

assert(i >= 0);

assert(i < num\_ranks);

index = i;

}

public CardRank(**char str**) {

index = determine\_index(str);

}

public boolean equals(CardRank s) {

return **s.index** == index;

}

**const char rank\_names [13] = {"Ace", "King", "Queen", "Jack", "10", "9", "8", "7", "6", "5", "4", "3", "2"};**

private int determine\_index(**char** str)

for (int i = 0; i < num\_ranks; ++i){

if (**str.equals(rank\_names[i])**)

return i;

**throw new RuntimeException**("Does not represent a card rank: " + str);

}

}

// Oliver Kullmann, 26.11.2009 (Swansea)

/\* Copyright 2009 Oliver Kullmann \*/

#include <iostream>

using std::cout;

class EvaluatedOutcome {

// For a given hand and an exchange request, an "evaluated outcome"

// is one of the possible hand-ranks which can arise after exchange,

// together with the probability that this hand-rank will be obtained.

public: EvaluatedOutcome(HandRank hr, double p) {

assert p >= 0;

assert p <= 1;

hand\_rank = hr;

prob = p;

}

public: HandRank; hand\_rank;

public: double prob;

public: boolean; equals(EvaluatedOutcome e) {

return **e.hand\_rank.equals(hand\_rank) && e.prob == prob**;

}

int main() {

cout << new EvaluatedOutcome(new HandRank(1), 0.9);

cout << hand\_rank + "\nProbability: " + prob;

}

}

// Oliver Kullmann, 4.12.2009 (Swansea)

/\* Copyright 2009 Oliver Kullmann \*/

#include <iostream>

class EvaluationResult {

public: EvaluationResult(EvaluatedOutcome[] E) {

assert E != null;

results = E;

length = **results.length**;

}

public: EvaluatedOutcome; get(int index) {

assert index >= 1;

assert index <= length;

return results[index-1];

}

**public: char** res = "";

for (int i = 0; i < length; ++i){

res += "\n" + (i+1) + ": " + results[i];

return res;

}

private: EvaluatedOutcome[]; results;

public: int length;

}

// Oliver Kullmann, 26.11.2009 (Swansea)

/\* Copyright 2009 Oliver Kullmann \*/

#include <iostream>

class Evaluation {

public: Evaluation(Hand h) {

assert h != null;

hand = h;

other\_cards = new Card[num\_other\_cards];

int[] deck = new int[**Card.num\_cards**];

for (int i = 0; i < **Card.num\_cards**; ++i){

deck[i] = i;

for (int i = 1; i <= **Hand.hand\_size**; ++i)

deck[**h.get(i).index()**] = -1;

int next\_index = 0;

for (int i = 0; i < **Card.num\_cards**; ++i)

if (deck[i] != -1)

other\_cards[next\_index++] = new Card(deck[i]);

}

}

// For an exchange-request e, compute the array of possible outcomes

// as hand-ranks, together with their probabilities, sorted by

// descending hand-ranks:

public: EvaluationResult; evaluate(ExchangeRequest e) {

if (**e.number\_cards** == 0) {

EvaluatedOutcome eo = new EvaluatedOutcome(new HandRank(hand), 1.0);

EvaluatedOutcome[] result = new EvaluatedOutcome[1];

result[0] = eo;

return new EvaluationResult(result);

}

int[] count\_outcomes = new int[HandRank.num\_hand\_ranks+1];

int num\_possibilities;

// run through all possibilities, and enter the result into count\_outcomes:

Card[] new\_hand = new Card[5];

for (int i = 0; i < **Hand.hand\_size**; ++i)

new\_hand[i] = hand.get(i+1);

if (**e.number\_cards** == 1) {

num\_possibilities = num\_other\_cards;

final int exchange\_index = **e.get\_index(1)**-1;

for (int i = 0; i < num\_other\_cards; ++i) {

new\_hand[exchange\_index] = other\_cards[i];

++count\_outcomes[**new HandRank((new Hand(new\_hand))).rank**];

}

}

else {

num\_possibilities = num\_other\_cards \* (num\_other\_cards-1);

int exchange\_index\_1 = **e.get\_index(1)**-1;

int exchange\_index\_2 = **e.get\_index(2)**-1;

for (int i = 0; i < num\_other\_cards; ++i) {

new\_hand[exchange\_index\_1] = other\_cards[i];

for (int j = i+1; j < num\_other\_cards; ++j) {

new\_hand[exchange\_index\_2] = other\_cards[j];

++count\_outcomes[new HandRank((**new Hand(new\_hand))).rank**];

}

}

}

int count\_different\_ranks = 0;

// determine the number of different ranks entered into count\_outcomes:

for (int i = 1; i <= **HandRank.num\_hand\_ranks**; ++i)

if (count\_outcomes[i] != 0) ++count\_different\_ranks;

// transfer the results into an array of evaluated outcomes:

EvaluatedOutcome[] result = new EvaluatedOutcome[count\_different\_ranks];

int next\_index = 0;

for (int i = 1; i <= **HandRank.num\_hand\_ranks**; ++i)

if (count\_outcomes[i] != 0)

result[next\_index++] = new EvaluatedOutcome(new HandRank(i), (double) count\_outcomes[i] / num\_possibilities);

return new EvaluationResult(result);//removing the brackets and/or new EvaluationResult doesn't help.

}

private :Hand; hand;

private :Card[]; other\_cards;

int main() {

using std::cout;

public: int num\_other\_cards = **Card.num\_cards - Hand.hand\_size**;

Bank b = new Bank(1);

Hand h = **b.orig\_hand(1)**;

Cout << h;

Evaluation E = new Evaluation(h);

{

cout << "\nNo exchange:";

ExchangeRequest e = new ExchangeRequest();

EvaluationResult R = **E.evaluate(e)**;

assert **R.length** == 1;

cout << R;

}

{

cout << "\nExchange last card:";

int[] ea = new int[1];

ea[0] = 5;

ExchangeRequest e = new ExchangeRequest(ea);

EvaluationResult R = **E.evaluate(e)**;

cout << R;

return 0;

}

}

}

// Oliver Kullmann, 26.11.2009 (Swansea)

/\* Copyright 2009 Oliver Kullmann \*/

#include <iostream>

class ExchangeRequest {

// For an object e of type ExchangeRequest, by e.number\_cards we obtain

// the number of cards to be exchanged, while by get\_index(i) for

// 1 <= i <= number\_cards we get the index of the i-th card to be exchanged.

/\*

For example assume that we want to exchange cards 2 and 5:

int[] ea = new int[2];

ea[0] = 2;

ea[1] = 5;

final ExchangeRequest e = new ExchangeRequest(ea);

assert e.number\_cards == 2;

assert e.get\_index(1) == 2;

assert e.get\_index(2) == 5;

\*/

public: int number\_cards;

public: ExchangeRequest() {

number\_cards = 0;

exchange\_indices = null;

}

public: ExchangeRequest(int e1) {

assert e1 >= 1;

assert e1 <= Hand.hand\_size;

number\_cards = 1;

exchange\_indices = new int[1];

exchange\_indices[0] = e1;

}

public: ExchangeRequest(int e1, int e2) {

assert e1 >= 1;

assert e1 <= **Hand.hand\_size**;

assert e2 >= 1;

assert e2 <= **Hand.hand\_size**;

assert e1 < e2;

number\_cards = 2;

exchange\_indices = new int[2];

exchange\_indices[0] = e1;

exchange\_indices[1] = e2;

}

public: ExchangeRequest; int[] exchange\_indices {

assert exchange\_indices\_ != null;

assert(**exchange\_indices\_.length** <= max\_exchanges);

for (int i = 0; i < **exchange\_indices\_.length**; ++i) {

assert(exchange\_indices\_[i] >= 1);

assert(exchange\_indices\_[i] <= **Hand.hand\_size**);

if (**exchange\_indices\_.length** == max\_exchanges)

assert exchange\_indices\_[0] < exchange\_indices\_[1];

exchange\_indices = exchange\_indices\_;

number\_cards = **exchange\_indices.length**;

}

public: int; get\_index(int i) {

assert i >= 1;

assert i <= number\_cards;

return exchange\_indices[i-1];

}

if (number\_cards == 0){

return "()";

if (number\_cards == 1){

return "(" + exchange\_indices[0] + ")";

return "(" + exchange\_indices[0] + "," + exchange\_indices[1] + ")";

}

}

public: boolean; **equals**(ExchangeRequest E) {

if (**E.number\_cards** != number\_cards) return false;

if (number\_cards == 0) return true;

if (number\_cards == 1) return **E.exchange\_indices**[0] == exchange\_indices[0];

return **E.exchange\_indices**[0] == exchange\_indices[0] && **E.exchange\_indices**[1] == exchange\_indices[1];

}

public: ExchangeRequest[] all\_requests() {

ExchangeRequest[] E = new ExchangeRequest[total\_num\_requests];

E[0] = new ExchangeRequest();

for (int i = 1; i <= **Hand.hand\_size**; ++i)

E[i] = new ExchangeRequest(i);

int next\_index = **Hand.hand\_size+1**;

for (int i = 1; i <= **Hand.hand\_size-1**; ++i)

for (int j =i+1; j <= **Hand.hand\_size**; ++j)

E[next\_index++] = new ExchangeRequest(i,j);

return E;

}

private: int[] exchange\_indices;

int main() {

using std::cout;

public: int max\_exchanges = 2;

public: int total\_num\_requests = 1 + 5 + 10;

final ExchangeRequest[] E = all\_requests();

**assert E.length** == total\_num\_requests;

for (int i = 0; i < total\_num\_requests; ++i)

cout << E[i];

}

}

// Oliver Kullmann, 26.11.2009 (Swansea)

/\* Copyright 2009 Oliver Kullmann \*/

#include <iostream>

class Hand {

// Provides poker hands in a standardised form, sorted by descending

// ranks.

// Three constructors are given, for reading five cards, an array of cards,

// or for reading from standard input.

// If the hand does not consist of five different cards, then an exception

// is thrown.

// Access to cards by get(i), where 1 <= i <= 5.

// Dependencies: In.java.

public: int hand\_size = 5;

public: int num\_hands = 2598960; // = binom(52,5)

// This constructor takes over ownership of c1,...,c5:

public: hand(Card c1, Card c2, Card c3, Card c4, Card c5) {

cards = new Card[hand\_size];

cards[0] = c1;

cards[1] = c2;

cards[2] = c3;

cards[3] = c4;

cards[4] = c5;

prepare\_hand();

assert h != null;

**assert h.length** == hand\_size;

cards = new Card[hand\_size];

for (int i = 0; i < hand\_size; ++i)

cards[i] = h[i];

prepare\_hand();

}

// Reading from an input stream:

public: hand; **In**; in(52) {

cards = new Card[hand\_size];

for (int i = 0; i < hand\_size); ++i {

CardRank rank = new CardRank(in.readString());

if (! **in.readstring().equals**("of"))

**throw new RuntimeException**("After the card-rank a string different from \"of\" has been found.");

Suit; suit = new Suit(**in.readString()**);

cards[i] = new Card(rank, suit);

}

prepare\_hand();

}

Card; get(int i) {

assert i >= 1;

assert i <= hand\_size;

return cards[i-1];

}

**char result = "";**

for (int i = 0; i < hand\_size; ++i) {

Card c = cards[i];

result += **c.rank**;

result += " of ";

result += **c.suit**;

result += "; ";

return result;

}

public: boolean; equals(Hand h) {

for (int i = 0; i < hand\_size; ++i)

if (! **h.cards[i].equals(cards[i])**)

return false;

return true;

}

private: Card[]; cards;

// Sorting by selection sort:

private: void sort\_by\_ranks(Card[] h) {

for (int i = 0; i < **h.length-1**; ++i) {

int index\_min = i;

for (int j = i+1; j < **h.length**; ++j)

if (**h[j].rank.index < h[index\_min].rank.index**)

index\_min = j;

if (index\_min != i) {

Card temp = h[i];

h[i] = h[index\_min];

h[index\_min] = temp;

}

}

}

/\* Remarks: If we wanted to use a sorting algorithm from the Java library,

then we needed some means to "tell" that algorithm the sorting criterions;

by what we learned in the module, yet we cannot provide such means.

By the above private method we can provide a specialised method,

tailored for our needs.

\*/

public: void check\_all\_different(Card[] h) {

for (int i = 0; i < hand\_size - 1; ++i)

if (**h[i].equals(h[i+1])**)

**throw new RuntimeException**("Two identical cards were found in a hand.");

}

**private: void prepare\_hand()** {

**sort\_by\_ranks(cards);**

**check\_all\_different(cards);**

}

}

// Oliver Kullmann, 26.11.2009 (Swansea)

/\* Copyright 2009 Oliver Kullmann \*/

#include <iostream>

class HandRank {

// Poker has 3614 ranks, where a rank is a number from 1 to 3614 assigned

// to a (Poker) hand such that a hand wins over another hand if and only if

// its rank is strictly lower (in case of equality thus we have a draw).

// The ranks are subdivided into 9 major ranks, where each is subdivided

// further into minor ranks.

// All three kinds of ranks are enumerated starting with value 1 for the

// highest possibility.

// The number of ranks up to a major rank:

// Access for a given major rank to the cumulated sizes via an array:

public: int cumulated\_size\_major\_ranks [10] = {0, cum\_num\_straight\_flushes, cum\_num\_four\_of\_a\_kinds, cum\_num\_full\_houses, cum\_num\_flushes, cum\_num\_straights, cum\_num\_three\_of\_a\_kinds, cum\_num\_two\_pairs, cum\_num\_one\_pairs, cum\_num\_high\_cards};

**public: char major\_hand\_rank\_names [9]** = {"Straight flush", "Four of a kind", "Full house", "Flush", "Straight", "Three of a kind", "Two pairs", "One pair", "High card"};

public: boolean valid\_hand\_rank( int r) {

return r >= 1 && r <= num\_hand\_ranks;

}

public: int major\_rank\_(int r) {

assert valid\_hand\_rank(r);

for (int i = 1; i <= num\_major\_hand\_ranks; ++i)

if (r <= cumulated\_num\_ranks[i]) return i;

return -1; // for the compiler; won't be executed for valid r

}

// Analyses a hand and returns an array with major-rank and minor-rank:

public: int[] hand\_rank(Hand h) {

int[] result = new int[2];

int[] rank\_count = new int[**CardRank.num\_ranks**];

for (int i = 1; i <= **Hand.hand\_size**; ++i)

++rank\_count[**h.get(i).rank.index**];

int[] count\_of\_counts = new int[**Suit.num\_suites+1**];

for (int i = 0; i < **CardRank.num\_ranks**; ++i)

++count\_of\_counts[rank\_count[i]];

if (count\_of\_counts[4] == 1) { // four of a kind

result[0] = four\_of\_a\_kind;

int rank\_quad = **h.get(2).rank.index**;

result[1] = rank\_quad + 1;

return result;

}

if (count\_of\_counts[3] == 1) {

int rank\_triple = **h.get(3).rank.index**;

result[1] = rank\_triple + 1;

if (count\_of\_counts[2] == 1) { // full house

result[0] = full\_house;

return result;

}

else { // three of a kind

result[0] = three\_of\_a\_kind;

return result;

}

}

if (count\_of\_counts[2] == 2) { // two pairs

result[0] = two\_pairs;

int[] ranks = new int[3];

transfer\_ranks(rank\_count, ranks);

int remaining\_ranks = **CardRank.num\_ranks - 2**;

result[1] = (lex\_order\_13(ranks[0],ranks[1])-1)\*remaining\_ranks+adjusted\_rank(ranks[0],ranks[1],ranks[2])+1;

return result;

}

if (count\_of\_counts[2] == 1) { // one pair

result[0] = one\_pair;

int[] ranks = new int[4];

transfer\_ranks(rank\_count, ranks);

int num\_remaining\_triples = 220; // = binom(13-1,3)

result[1] = ranks[0]\*num\_remaining\_triples+lex\_order\_12(adjusted\_rank(ranks[0],ranks[1]),adjusted\_rank(ranks[0],ranks[2]),adjusted\_rank(ranks[0],ranks[3]));

return result;

}

if (is\_straight(h)) {

if (**h.get(1).rank.index == CardRank.ace && h.get(2).rank.index == CardRank.five**)

result[1] = **CardRank.five** + 1;

else

result[1] = **h.get(1).rank.index** + 1;

if (is\_flush(h))

result[0] = straight\_flush;

else

result[0] = straight;

return result;

}

else {

int[] ranks = new int[5];

transfer\_ranks(rank\_count, ranks);

result[1] = lex\_order\_13(ranks[0],ranks[1],ranks[2],ranks[3],ranks[4]);

result[1] -= **h.get(1).rank.index+1**;

if (**h.get(1).rank.index != CardRank.ace**) // subtracting the low-ace-case

--result[1];

if (is\_flush(h))

result[0] = flush;

else

result[0] = high\_card;

return result;

}

}

HandRank(int r) {

assert valid\_hand\_rank(r);

rank = r;

major\_rank = major\_rank\_(rank);

minor\_rank = r - cumulated\_num\_ranks[major\_rank-1];

}

HandRank(Hand h) {

int[] hr = hand\_rank(h);

major\_rank = hr[0];

minor\_rank = hr[1];

rank = cumulated\_num\_ranks[major\_rank-1] + minor\_rank;

}

public: int rank;

public: int major\_rank;

public: int minor\_rank;

**public: int full\_rank()** {

return "Major rank: " + major\_hand\_rank\_names[major\_rank-1] + "\nMinor rank: " + minor\_rank + "; total rank: " + rank;

}

public: boolean; equals( HandRank hr) {

return **hr.rank == rank**;

}

// The probability that a (strictly) better hand than the given hand

// occurs for a random hand:

public: double; cumulated\_probability() {

return (double) cumulated\_count() / **Hand.num\_hands**;

}

// The number of (strictly) better hands (than the given hand):

public: int cumulated\_count() {

return cumulated\_size\_major\_ranks[major\_rank-1] + (minor\_rank-1) \* size\_ranks[major\_rank];

}

public: boolean; is\_flush( Hand h) {

Suit first = **h.get(1).suit**;

for (int i = 2; i <= 5; ++i)

if (! **h.get(i).suit.equals(first)**)

return false;

return true;

}

public: boolean; is\_straight( Hand h) {

if (**h.get(1).rank.index == CardRank.ace && h.get(2).rank.index == CardRank.five && h.get(5).rank.index == CardRank.two**)

return true;

else if (**h.get(1).rank.index + 5 - 1 == h.get(5).rank.index**)

return true;

else

return false;

}

// Transfer the ranks from rank\_count to ranks, where rank\_count[j] > 0

// means that rank j is present, and will be entered into the ordered list

// "ranks" of ranks, where pairs come first (no rank occurs more than

// twice):

private void transfer\_ranks( int[] rank\_count, int[] ranks) {

int i = 0;

for (int j = 0; j < **CardRank.num\_ranks**; ++j)

if (rank\_count[j] == 2)

ranks[i++] = j;

for (int j = 0; j < **CardRank.num\_ranks**; ++j)

if (rank\_count[j] == 1)

ranks[i++] = j;

}

// Determine the adjusted rank of "rank" when p1, p2 are not taken into

// account:

private: int adjusted\_rank( int p1, int rank) {

if (rank <= p1) return rank;

return rank - 1;

}

private: int adjusted\_rank( int p1, int p2, int rank) {

assert p1 < p2;

if (rank <= p1) return rank;

if (rank <= p2) return rank - 1;

return rank - 2;

}

// Functions for ranking subsets S of {0,1,...,12} resp. {0,1,...,11}

// for set-sizes 2,3,5; the elements of S are given by x1 < ... < x5:

private: int lex\_order\_13( int x1, int x2) {

return 66-(11-x1)\*(12-x1)/2+x2;

}

private: int lex\_order\_12( int x1, int x2, int x3) {

return 209+(-(9-x1)\*(10-x1)\*(11-x1))/6-(10-x2)\*(11-x2)/2+x3;

}

private: int lex\_order\_13( int x1, int x2, int x3, int x4, int x5) {

return 1275+(-(8-x1)\*(9-x1)\*(10-x1)\*(11-x1)\*(12-x1))/120+(-(9-x2)\*(10-x2)\*(11-x2)\*(12-x2))/24+(-(10-x3)\*(11-x3)\*(12-x3))/6-(11-x4)\*(12-x4)/2+x5;

}

// Tests (run by "java -ea HandRank", enabling assertions):

int main() {

//the below was initially at the top of the code

public: int num\_major\_hand\_ranks = 9;

public: int straight\_flush = 1;

public: int four\_of\_a\_kind = 2;

public: int full\_house = 3;

public: int flush = 4;

public: int straight = 5;

public: int three\_of\_a\_kind = 6;

public: int two\_pairs = 7;

public: int one\_pair = 8;

public: int high\_card = 9;

// The number of minor ranks in a major rank:

public: int num\_straight\_flush\_ranks = 10;

public: int num\_four\_of\_a\_kind\_ranks = 13;

public: int num\_full\_house\_ranks = 13;

public: int num\_flush\_ranks = 1277;

public: int num\_straight\_ranks = 10;

public: int num\_three\_of\_a\_kind\_ranks = 13;

public: int num\_two\_pairs\_ranks = 858;

public: int num\_one\_pair\_ranks = 2860;

public: int num\_high\_card\_ranks = 1277;

public: int size\_straight\_flush\_rank = 4;

public: int size\_four\_of\_a\_kind\_rank = 48;

public: int size\_full\_house\_rank = 288;

public: int size\_flush\_rank = 4;

public: int size\_straight\_rank = 1020;

public: int size\_three\_of\_a\_kind\_rank = 4224;

public: int size\_two\_pairs\_rank = 144;

public: int size\_one\_pair\_rank = 384;

public: int size\_high\_card\_rank = 1020;

public: int cum\_straight\_flush\_ranks = num\_straight\_flush\_ranks;

public: int cum\_four\_of\_a\_kind\_ranks = cum\_straight\_flush\_ranks + num\_four\_of\_a\_kind\_ranks;

public: int cum\_full\_house\_ranks = cum\_four\_of\_a\_kind\_ranks + num\_full\_house\_ranks;

public: int cum\_flush\_ranks = cum\_full\_house\_ranks + num\_flush\_ranks;

public: int cum\_straight\_ranks = cum\_flush\_ranks + num\_straight\_ranks;

public: int cum\_three\_of\_a\_kind\_ranks = cum\_straight\_ranks + num\_three\_of\_a\_kind\_ranks;

public: int cum\_two\_pairs\_ranks = cum\_three\_of\_a\_kind\_ranks + num\_two\_pairs\_ranks;

public: int cum\_one\_pair\_ranks = cum\_two\_pairs\_ranks + num\_one\_pair\_ranks;

public: int cum\_high\_card\_ranks = cum\_one\_pair\_ranks + num\_high\_card\_ranks;

// Access for a given major rank to the cumulated ranks via an array:

public: int[] cumulated\_num\_ranks = {0, cum\_straight\_flush\_ranks, cum\_four\_of\_a\_kind\_ranks, cum\_full\_house\_ranks, cum\_flush\_ranks, cum\_straight\_ranks, cum\_three\_of\_a\_kind\_ranks, cum\_two\_pairs\_ranks, cum\_one\_pair\_ranks, cum\_high\_card\_ranks};

public: int num\_hand\_ranks = cum\_high\_card\_ranks; // = 6331

// For a given rank, how many hands of that rank are there (this depends

// only on the major rank):

// Access for a given rank to the sizes via an array (through the associated

// major rank):

public: int[] size\_ranks = {0, size\_straight\_flush\_rank, size\_four\_of\_a\_kind\_rank, size\_full\_house\_rank, size\_flush\_rank, size\_straight\_rank, size\_three\_of\_a\_kind\_rank, size\_two\_pairs\_rank, size\_one\_pair\_rank, size\_high\_card\_rank};

// The number of hands of a given major rank:

public: int num\_straight\_flushes = size\_straight\_flush\_rank \* num\_straight\_flush\_ranks;

public: int num\_four\_of\_a\_kinds = size\_four\_of\_a\_kind\_rank \* num\_four\_of\_a\_kind\_ranks;

public: int num\_full\_houses = size\_full\_house\_rank \* num\_full\_house\_ranks;

public: int num\_flushes = size\_flush\_rank \* num\_flush\_ranks;

public: int num\_straights = size\_straight\_rank \* num\_straight\_ranks;

public: int num\_three\_of\_a\_kinds = size\_three\_of\_a\_kind\_rank \* num\_three\_of\_a\_kind\_ranks;

public: int num\_two\_pairs = size\_two\_pairs\_rank \* num\_two\_pairs\_ranks;

public: int num\_one\_pairs = size\_one\_pair\_rank \* num\_one\_pair\_ranks;

public: int num\_high\_cards = size\_high\_card\_rank \* num\_high\_card\_ranks;

// Access for a given major rank to its size via an array:

public: int[] size\_major\_ranks = {0, num\_straight\_flushes, num\_four\_of\_a\_kinds, num\_full\_houses, num\_flushes, num\_straights, num\_three\_of\_a\_kinds, num\_two\_pairs, num\_one\_pairs, num\_high\_cards};

// The number of hands up to a given major rank:

public: int cum\_num\_straight\_flushes = num\_straight\_flushes;

public: int cum\_num\_four\_of\_a\_kinds = cum\_num\_straight\_flushes + num\_four\_of\_a\_kinds;

public: int cum\_num\_full\_houses = cum\_num\_four\_of\_a\_kinds + num\_full\_houses;

public: int cum\_num\_flushes = cum\_num\_full\_houses + num\_flushes;

public: int cum\_num\_straights = cum\_num\_flushes + num\_straights;

public: int cum\_num\_three\_of\_a\_kinds = cum\_num\_straights + num\_three\_of\_a\_kinds;

public: int cum\_num\_two\_pairs = cum\_num\_three\_of\_a\_kinds + num\_two\_pairs;

public: int cum\_num\_one\_pairs = cum\_num\_two\_pairs + num\_one\_pairs;

public: int cum\_num\_high\_cards = cum\_num\_one\_pairs + num\_high\_cards;

//-------------------------------------------------------------------was above.

assert num\_hand\_ranks == 6331;

assert num\_straight\_flushes == 40;

assert num\_four\_of\_a\_kinds == 624;

assert num\_full\_houses == 3744;

assert num\_flushes == 5108;

assert num\_straights == 10200;

assert num\_three\_of\_a\_kinds == 54912;

assert num\_two\_pairs == 123552;

assert num\_one\_pairs == 1098240;

assert num\_high\_cards == 1302540;

assert cum\_num\_high\_cards == **Hand.num\_hands**;

// Testing the various functions for ranking subsets according to

// lexicographical order:

assert lex\_order\_13(0,1) == 1;

assert lex\_order\_13(11,12) == 78;

assert lex\_order\_12(0,1,2) == 1;

assert lex\_order\_12(9,10,11) == 220;

assert lex\_order\_13(0,1,2,3,4) == 1;

assert lex\_order\_13(8,9,10,11,12) == 1277 + 10;

{

Hand h = new Hand(new Card(0), new Card(1), new Card(14), new Card(2), new Card(15));

HandRank hr = new HandRank(h);

**assert hr.major\_rank** == two\_pairs;

**assert hr.minor\_rank** == 133;

}

{

Hand h = new Hand(new Card(3), new Card(1), new Card(14), new Card(2), new Card(15));

HandRank hr = new HandRank(h);

**assert hr.major\_rank** == two\_pairs;

**assert hr.minor\_rank** == 134;

}

{

Hand h = new Hand(new Card(0), new Card(13), new Card(2), new Card(15), new Card(1));

HandRank hr = new HandRank(h);

**assert hr.major\_rank** == two\_pairs;

**assert hr.minor\_rank** == 12;

}

{

Hand h = new Hand(new Card(0), new Card(1), new Card(14), new Card(2), new Card(3));

HandRank hr = new HandRank(h);

**assert hr.major\_rank** == one\_pair;

**assert hr.minor\_rank** == 221;

}

// Running through all possible hands, and detemining the ranks:

int[] statistics\_ranks = new int[num\_hand\_ranks+1];

int[] statistics\_major\_ranks = new int[num\_major\_hand\_ranks+1];

for (int c1 = 0; c1 < **Card.num\_cards** - 4; ++c1) {

Card C1 = new Card(c1);

for (int c2 = c1+1; c2 < **Card.num\_cards** - 3; ++c2) {

Card C2 = new Card(c2);

for (int c3 = c2+1; c3 < **Card.num\_cards** - 2; ++c3) {

Card C3 = new Card(c3);

for (int c4 = c3+1; c4 < **Card.num\_cards** - 1; ++c4) {

Card C4 = new Card(c4);

for (int c5 = c4+1; c5 < **Card.num\_cards**; ++c5) {

Card C5 = new Card(c5);

Hand h = new Hand(C1,C2,C3,C4,C5);

HandRank hr = new HandRank(h);

++statistics\_ranks[**hr.rank**];

++statistics\_major\_ranks[**hr.major\_rank**];

}

}

}

}

}

statistics\_major\_ranks[0] = 0;

for (int i = 0; i <= num\_major\_hand\_ranks; ++i)

assert statistics\_major\_ranks[i] == size\_major\_ranks[i];

for (int rank = 1; rank <= num\_hand\_ranks; ++rank) {

HandRank hr = new HandRank(rank);

assert statistics\_ranks[rank] == size\_ranks[**hr.major\_rank**];

}

}

}

// Oliver Kullmann, 26.11.2009 (Swansea)

/\* Copyright 2009 Oliver Kullmann \*/

#include <iostream>

class TwoPlayers {

// Plays a single game between two players, using all available strategies

// from the Strategy class, and returns an object of type Storage with

// entries from {-1,0,+1}, where an entry -1 means player 2 wins, +1 means

// player 1 wins, and 0 means draw:

public: int Storage; **play()** {

Storage R = new Storage(**Strategy.number\_strategies**);

Bank b = new Bank(2);

ExchangeRequest[] ep1 = new ExchangeRequest[**Strategy.number\_strategies-1**];

{

Strategy p1 = new Strategy(**b.orig\_hand(1)**);

for (int s1 = 1; s1 < **Strategy.number\_strategies**; ++s1)

ep1[s1-1] = **p1.exchange(s1)**;

}

ExchangeRequest[] ep2 = new ExchangeRequest[**Strategy.number\_strategies-1**];

{

Strategy p2 = new Strategy(**b.orig\_hand(2)**);

for (int s2 = 2; s2 <= **Strategy.number\_strategies**; ++s2)

ep2[s2-2] = **p2.exchange**(s2);

}

ExchangeRequest[] E = new ExchangeRequest[2];

for (int s1 = 1; s1 < **Strategy.number\_strategies**; ++s1)

for (int s2 = s1+1; s2 <= **Strategy.number\_strategies**; ++s2) {

E[0] = ep1[s1-1];

E[1] = ep2[s2-2];

**b.perform\_exchanges(E)**;

HandRank hr1 = new HandRank(**b.new\_hand(1)**);

HandRank hr2 = new HandRank(**b.new\_hand(2)**);

if (**hr1.rank < hr2.rank**)

**R.set(s1,s2,+1);**

else if (**hr2.rank < hr1.rank**)

**R.set(s1,s2,-1);**

else

**R.set(s1,s2,0);**

}

return R;

}

int main() {

using std::cout;

Storage S = **TwoPlayers.play()**;

cout << S;

}

}

// Oliver Kullmann, 26.11.2009 (Swansea)

/\* Copyright 2009 Oliver Kullmann \*/

#include <iostream>

class Tournament {

// Read the number N of rounds from the command-line, and for each

// pair 1 <= i < j <= Strategy.number\_strategies of strategies from Strategy,

// play N rounds, showing the difference of the number of games won by

// strategy i minus the number of games won by strategy j.

int main() {

using std::cout;

if (**args.length** == 0) {

cout << "ERROR[Tournament]: One parameter is needed, the number N of games.";

return;

}

long N = **Long.parseLong**(args[0]);

Storage S = new Storage(**Strategy.number\_strategies**);

for (long t = 1; t <= N; ++t) {

Storage R = **TwoPlayers.play()**;

for (int i = 1; i < **Strategy.number\_strategies**; ++i)

for (int j = i+1; j <= **Strategy.number\_strategies**; ++j)

**S.set**(i,j, **S.get**(i,j)+ **R.get**(i,j));

}

cout << S;

}

}

// Oliver Kullmann, 26.11.2009 (Swansea)

/\* Copyright 2009 Oliver Kullmann \*/

//need to find out how to call strings.

#include <iostream>

class Suit {

// Wrapper class around suites as integer indices.

public: int num\_suites = 4;

public: int clubs = 0;

public: int spades = 1;

public: int hearts = 2;

public: int diamonds = 3;

public: int index;

public: Suit( int i) {

assert(i >= 0);

assert(i < num\_suites);

index = i;

}

public: Suit(**char** str) {

index = determine\_index(str);

}

**public: boolean; equals()** {

return suit\_names[index];

}

**public: boolean; equals(Suit s)** {

return **s.index** == index;

}

**const char suit\_names [4] = {"Clubs", "Spades", "Hearts", "Diamonds" };**

private: int determine\_index(**char** str) {

for (int i = 0; i < num\_suites; ++i)

if (**str.equals**(suit\_names[i]))

return i;

**throw new RuntimeException**("Does not represent a suit: " + str);

}

}

// Oliver Kullmann, 26.11.2009 (Swansea)

/\* Copyright 2009 Oliver Kullmann \*/

#include <iostream>

class Strategy {

// For an object s of type Strategy by s.exchange(i) we obtain

// the exchange request for the strategy with index i.

public: int number\_strategies = 5; // to be updated

public: Strategy(Hand h\_) {

h = h\_;

}

public: ExchangeRequest; exchange( int index\_strategy) {

assert valid\_index(index\_strategy);

assert index\_strategy <= number\_strategies;

switch (index\_strategy) {

case 1 : return no\_exchange();

case 2 : return always\_first\_two();

case 3 : return always\_last();

case 4 : return always\_last\_two();

case 5 : return minimise\_expected\_number\_better\_hands();

case 6 : return risky();

case 7 : return cautious();

}

return **no\_exchange()**; // for the compiler; does not occur for valid indices

}

public: boolean; valid\_index( int i) {

return i >= 1 && i <= number\_strategies;

}

private: Hand; h;

private: ExchangeRequest; **no\_exchange()** {

return new **ExchangeRequest()**;

}

private: ExchangeRequest; always\_first\_two() {

int[] e = new int[2];

e[0] = 1;

e[1] = 2;

return new ExchangeRequest(e);

}

private: ExchangeRequest; **always\_last()** {

int[] e = new int[1];

e[0] = 5;

return new ExchangeRequest(e);

}

private: ExchangeRequest; **always\_last\_two()** {

int[] e = new int[2];

e[0] = 4;

e[1] = 5;

return new ExchangeRequest(e);

}

private: ExchangeRequest; minimise\_expected\_number\_better\_hands() {

Evaluation eval = new Evaluation(h);

ExchangeRequest[] all\_requests = **ExchangeRequest.all\_requests()**;

ExchangeRequest best\_exchange\_request = null;

double best\_expected\_value = **Double.POSITIVE\_INFINITY**;

for (int i = 0; i < **all\_requests.length**; ++i) {

ExchangeRequest new\_exchange\_request = all\_requests[i];

double new\_expected\_value = expected\_number\_better\_hands(**eval.evaluate(new\_exchange\_request)**);

if (new\_expected\_value < best\_expected\_value) {

best\_expected\_value = new\_expected\_value;

best\_exchange\_request = new\_exchange\_request;

}

}

return best\_exchange\_request;

}

private: double expected\_number\_better\_hands( EvaluationResult E) {

double sum = 0;

for (int i = 1; i <= **E.length**; ++i) {

EvaluatedOutcome e = **E.get(i)**;

sum += **e.hand\_rank.cumulated\_count() \* e.prob**;

}

return sum;

}

**// Go for the best hand achievable:**

**private: ExchangeRequest; risky() {**

**// XXX**

**return new ExchangeRequest(); // temporarily**

**}**

**// Choose only amongst choices which can not impair the original hand:**

**private: ExchangeRequest; cautious() {**

**// XXX**

**return new ExchangeRequest(); // temporarily**

**}**

// Possibly further strategies to be implemented XXX

// Tests:

int main() {

using std::cout;

{

Hand h = new Hand(new Card(0), new Card(1), new Card(2), new Card(3), new Card(25));

Strategy s = new Strategy(h);

ExchangeRequest e = **s.exchange**(5);

**assert** (**e.equals**(new ExchangeRequest(4,5)));

}

Bank b = new Bank(1);

Hand h = **b.orig\_hand**(1);

Strategy s = new Strategy(h);

cout << h;

for (int i = 1; i <= number\_strategies; ++i) {

cout << i + ": " + **s.exchange**(i);

}

}

}

// Oliver Kullmann, 26.11.2009 (Swansea)

/\* Copyright 2009 Oliver Kullmann \*/

#include <iostream>

class Storage {

// Provides storage for the results of all (different) strategies against

// each other (for a fixed deck), usable for the single results as well as

// for the statistics.

// Remark: One could call this class also "UpperTriangularMatrix".

/\*

Via Storage(5) for example storage for 5\*4/2 = 10 integers is

provided. Via set(i,j,x) then for the pairs 1 <= i < j <= 5

the value x is stored, and retrieved by get(i,j).

\*/

public: int number\_strategies;

public: int number\_results;

public: Storage(int num\_strategies) {

assert num\_strategies >= 0;

number\_strategies = num\_strategies;

number\_results = (number\_strategies \* (number\_strategies-1)) / 2;

results = new int[number\_results];

}

public: int get( int i, int j) {

return results[index(i,j)];

}

public: void set( int i, int j, int x) {

results[index(i,j)] = x;

}

**char result** = "";

for (int i = 1; i < number\_strategies; ++i) {

for (int j = i+1; j <= number\_strategies; ++j)

result += get(i,j) + " ";

result += "\n";

return result;

}

public: boolean; equals( Storage S) {

if (**S.number\_strategies** != number\_strategies)

return false;

for (int i = 1; i < number\_strategies; ++i)

for (int j = i+1; j <= number\_strategies; ++j)

if (**S.get**(i,j) != get(i,j))

return false;

return true;

}

private: int[] results;

// Using colexicographical order for linearisation:

private: int index(int i, int j) {

assert i >= 1;

assert i <= number\_strategies;

assert j >= 1;

assert j <= number\_strategies;

assert i < j;

return ((j-2)\*(j-1))/2+i-1;

}

// Tests:

int main() {

Storage S = new Storage(5);

assert **S.number\_strategies** == 5;

assert **S.number\_results** == 10;

assert **S.equals(S)**;

for (int i = 1; i < 5; ++i)

for (int j = i+1; j <= 5; ++j)

**S.set**(i,j,i\*j);

**assert S.equals**(S);

for (int i = 1; i < 5; ++i)

for (int j = i+1; j <= 5; ++j)

**assert(S.get**(i,j) == i\*j);

Storage S2 = new Storage(5);

**assert ! S.equals**(S2);

for (int i = 1; i < 5; ++i)

for (int j = i+1; j <= 5; ++j)

**S2.set**(i,j,i\*j);

**assert S.equals**(S2);

**S2.set**(1,2,77);

**assert ! S.equals**(S2);

}

}

// Oliver Kullmann, 26.11.2009 (Swansea)

/\* Copyright 2009 Oliver Kullmann \*/

#include <iostream>

class PokerExtended {

// Simulating the behaviour of the Poker-program from the first

// coursework (with additional information when evaluating a single hand):

int main() {

using std::cout;

if (**args.length** == 0) {

cout << "ERROR[Poker]: One argument N is required.";

return;

}

int N = **Integer.parseInt**(args[0]);

if (N < 0) {

cout << "ERROR[Poker]: N < 0.";

return;

}

if (N == 0) {

Hand h = new Hand(new In());

HandRank hr = new HandRank(h);

cout << hr;

cout << "The number of (strictly) better hands: " + **hr.cumulated\_count()**;

cout << "The probability of a (strictly) better hand: " + (100 \* **hr.cumulated\_probability()**) + "%";

cout << "Analysis of the exchange possibilities:";

ExchangeRequest[] all = **ExchangeRequest.all\_requests()**;

Evaluation eval = new Evaluation(h);

for (int i = 0; i < all.length; ++i) {

cout << "Exchange " + all[i] + " yields:";

EvaluationResult result = eval.evaluate(all[i]);

cout << "Best: ";

cout << **result.get**(1);

cout << "Worst: ";

cout << **result.get(result.length)**;

}

}

else {

int[] counts = new int[**HandRank.num\_major\_hand\_ranks+1**];

for (int i = 1; i <= N; ++i) {

Bank b = new Bank(1);

HandRank hr = new HandRank(**b.orig\_hand(1)**);

++counts[**hr.major\_rank**];

}

for (int r = 1; r <= **HandRank.num\_major\_hand\_ranks**; ++r)

cout << **HandRank.major\_hand\_rank\_names**[r-1] + ": " + (double) (counts[r]) / N \* 100 + "%";

}

}

}

// Oliver Kullmann, 26.11.2009 (Swansea)

/\* Copyright 2009 Oliver Kullmann \*/

#include <iostream>

class Poker {

// Simulating the behaviour of the Poker-program from the first

// coursework (with additional information when evaluating a single hand):

int main() {

using std::cout;

if (**args.length** == 0) {

cout << "ERROR[Poker]: One argument N is required.";

return;

}

int N = **Integer.parseInt**(args[0]);

if (N < 0) {

cout << "ERROR[Poker]: N < 0.";

return;

}

if (N == 0) {

Hand h = new Hand(new In());

HandRank hr = new HandRank(h);

cout << hr;

cout << "The number of (strictly) better hands: " + **hr.cumulated\_count()**;

cout << "The probability of a (strictly) better hand: " + (100 \* **hr.cumulated\_probability()**) + "%";

}

else {

int[] counts = new int[**HandRank.num\_major\_hand\_ranks+1**];

for (int i = 1; i <= N; ++i) {

Bank b = new Bank(1);

HandRank hr = new HandRank(**b.orig\_hand(1)**);

++counts[**hr.major\_rank**];

}

for (int r = 1; r <= **HandRank.num\_major\_hand\_ranks**; ++r)

cout << **HandRank.major\_hand\_rank\_names[r-1]** + ": " + (double) (counts[r]) / N \* 100 + "%";

}

}

}

// Oliver Kullmann, 5.11.2009 (Swansea)

/\*

To be compiled by

javac Poker.java

(needs StdIn.java), and to be run by

java Poker N

where N >= 0 is an integer. For N = 0, a hand of five cards is read from

standard input, and its hand rank (from straight flush to high card) is

output, while for N > 0 this number of random hands are drawn, and

the statistics on the relative frequency of the nine hand ranks is

output.

The format of a card to to be read is for example "King of Clubs" (that is,

card-rank "of" suit), where space-symbols are irrelevant (except of the

separating space-symbols). See below for the exact spelling of card-ranks

and suites.

\*/

#include <iostream>

#include <stdio.h>

#include "String.h"

using namespace std;

class Poker {

// Suites are represented by integers 0,...,3,

// card ranks by integers 0,...,12, and cards by integers 0,...,51.

// And hand ranks are represented by integer 1,...,9 (from highest

// to lowest).

static const int num\_suites = 4;

static const int num\_ranks = 13;

static const int num\_cards = num\_suites \* num\_ranks;

static const int hand\_size = 5;

static const int num\_hand\_ranks = 9;

// The integer-representation of suites and ranks are given as indices of

// the following arrays:

//the arrays have been declared in the main function below

/\*static const int MaxLength = 80;

char String[4] ;suit\_names = "Clubs", "Diamonds", "Hearts", "Spades"

char String[13] rank\_names = "2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack", "Queen", "King", "Ace"

// The string representation of hand ranks:

char String[9] hand\_rank\_names = "Straight flush", "Four of a kind", "Full house", "Flush", "Straight", "Three of a kind", "Two pairs", "One pair", "High card"

// Explicit constants for the 9 hand ranks:\*/

static const int straight\_flush = 1;

static const int four\_of\_a\_kind = 2;

static const int full\_house = 3;

static const int flush = 4;

static const int straight = 5;

static const int three\_of\_a\_kind = 6;

static const int two\_pairs = 7;

static const int one\_pair = 8;

static const int high\_card = 9;

// Converts a string into a suit; returns -1, if the string doesn't

// represent a suit:

public: int parse\_suit(String s) {

for (int i = 0; i < num\_suites; ++i)

if (**s.equals(suit\_names[i])**) return i;

return -1;

}

// Converts a string into a card-rank; returns -1, if the string doesn't

// represent a card-rank:

public: int parse\_rank(**String s**) {

for (int i = 0; i < num\_ranks; ++i)

if (**s.equals(rank\_names[i])**) return i;

return -1;

}

// For a card, compute its suit:

public: int suit( int card) {

return card / num\_ranks;

}

// For a card, compute its rank:

public: int rank( int card) {

return card % num\_ranks;

}

// Given card-rank and suit, compute the corresponding card:

public: int card( int rank, int suit) {

return suit \* num\_ranks + rank;

}

// Read a hand (and array of cards) from standard-input; returns null

// if some parsing error occurs:

//will need to change the StdIn parts!!!

public: int[] read\_hand() {

int[] hand = new int[hand\_size];

for (int i = 0; i < hand\_size; i++) {

int rank = parse\_rank(StdIn.readString());

if (rank == -1) return null;

if (! **StdIn.readString().equals("of")**) return null;

int suit = parse\_suit(**StdIn.readString()**);

if (suit == -1) return null;

hand[i] = card(rank, suit);

}

return hand;

}

// Creates a random hand:

public: int[] random\_hand() {

int[] hand = new int[hand\_size];

int[] cards = new int[num\_cards];

for (int i = 0; i < num\_cards; ++i)

cards[i] = i;

for (int i = 0; i < hand\_size; ++i) {

int random\_index = i + (int) (**Math.random()** \* (num\_cards - i));

// i <= random\_index < num\_cards - i

hand[i] = cards[random\_index];

cards[random\_index] = cards[i];

}

return hand;

}

// Check that the cards of a hand are really all different:

private: boolean; check\_all\_different( int[] hand) {

for (int i = 0; i < hand\_size; ++i)

for (int j = i+1; j < hand\_size; ++j)

if (hand[i] == hand[j]) return false;

return true;

}

// Check whether a hand is a flush:

private: boolean; is\_flush( int[] hand) {

int first\_suit = suit(hand[0]);

for (int i = 1; i < hand\_size; ++i)

if (suit(hand[i]) != first\_suit) return false;

return true;

}

// Check whether a hand is a straight:

private: boolean; is\_straight( int[] hand) {

boolean[] ranks = new boolean[num\_ranks+1];

// shifting ranks 0..12 to 1..13, and adding new rank 0 for card "1"

for (int i = 0; i < hand\_size; ++i)

ranks[rank(hand[i])+1] = true;

if (ranks[num\_ranks] & ranks[1]) ranks[0] = true;

int first = 0;

while (! ranks[first]) ++first;

int num\_remains = hand\_size - 1;

if (first + num\_remains > num\_ranks) return false;

for (int i = first + 1; i <= first + num\_remains; ++i)

if (! ranks[i]) return false;

return true;

}

/\* Remark: An alternative algorithm is to first sort the hand by

ranks, using the sort-algorithm from the Java-library. This would

yield more compact code, however in this module we use only elements

from the Java-library as discussed in the lectures.

\*/

// Determine the hand-rank of a hand:

public: int hand\_rank(int[] hand {

int[] rank\_count = new int[num\_ranks];

for (int i = 0; i < hand\_size; ++i)

++rank\_count[rank(hand[i])];

int[] count\_of\_counts = new int[num\_suites+1];

for (int i = 0; i < num\_ranks; ++i)

++count\_of\_counts[rank\_count[i]];

if (count\_of\_counts[4] == 1) return four\_of\_a\_kind;

if (count\_of\_counts[3] == 1)

if (count\_of\_counts[2] == 1) return full\_house;

else return three\_of\_a\_kind;

if (count\_of\_counts[2] == 2) return two\_pairs;

if (count\_of\_counts[2] == 1) return one\_pair;

boolean is\_flush = is\_flush(hand);

boolean is\_straight = is\_straight(hand);

if (is\_flush && is\_straight) return straight\_flush;

else if (is\_flush) return flush;

else if (is\_straight) return straight;

else return high\_card;

}

int main()

{

static const int MaxLength = 80;

// Explicit constants for the 9 hand ranks: **char String[4] ;suit\_names = "Clubs", "Diamonds", "Hearts", "Spades";**

**char String[13] rank\_names = "2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack", "Queen", "King", "Ace";**

// The string representation of hand ranks:

**char String[9] hand\_rank\_names = "Straight flush", "Four of a kind", "Full house", "Flush", "Straight", "Three of a kind", "Two pairs", "One pair", "High card";**

// the above variables were above at the beginning of the code

if (**args.length** == 0) {

cout << "ERROR[Poker]: One argument N is required.";

return 0;

}

int N = **Integer.parseInt**(args[0]);

if (N < 0) {

cout << "ERROR[Poker]: N < 0.";

return 0;

}

if (N == 0) {

int[] hand = read\_hand();

if (hand == null) {

cout << "ERROR[Poker]: Incorrect hand of cards.";

return 0;

}

if (! check\_all\_different(hand)) {

cout << "ERROR[Poker]: Two cards coincide.";

return 0;

}

cout << "Hand rank = " + hand\_rank\_names[hand\_rank(hand)-1];

}

else {

int[] counts = new int[num\_hand\_ranks+1];

for (int i = 1; i <= N; ++i)

++counts[hand\_rank(random\_hand())];

for (int hand\_rank = 1; hand\_rank <= num\_hand\_ranks; ++hand\_rank)

cout << (hand\_rank\_names[hand\_rank-1] + ": " + (double) (counts[hand\_rank]) / N \* 100 + "%");

}

}

}