Winston Wang

Mr. Kuszmaul

AP Computer Science

19 October 2014

Chapter 15 Homework

**Exercise 1:**

The changed method for merge() is below:

**public** Deck merge(Deck d2)

{

// create the new deck

Deck result = **new** Deck (**this**.cards.length + d2.cards.length);

**int** choice; // records the winner (1 means d1, 2 means d2)

**int** i = 0; // traverses the first input deck

**int** j = 0; // traverses the second input deck

// k traverses the new (merged) deck

**for** (**int** k = 0; k < result.cards.length; k++) {

choice = 1;

// if d1 is empty, d2 wins; if d2 is empty, d1 wins; otherwise,

// compare the two cards

**if** (i == **this**.cards.length)

choice = 2;

**else** **if** (j == d2.cards.length)

choice = 1;

**else** **if** (**this**.cards[i].compareTo(d2.cards[j]) > 0)

choice = 2;

// make the new deck refer to the winner card

**if** (choice == 1) {

result.cards[k] = **this**.cards[i]; i++;

} **else** {

result.cards[k] = d2.cards[j]; j++;

}

}

**return** result;

}  
The new method for the mergeSort() is below:

/\*

\* Sort the Deck using merge sort.

\*/

**public** Deck mergeSort() {

**if** (cards.length < 2) {

**return** **this**;

}

**int** mid = (cards.length-1) / 2;

// divide the deck roughly in half

Deck d1 = subdeck(0, mid);

Deck d2 = subdeck(mid+1, cards.length-1);

// sort the halves

d1 = d1.mergeSort();

d2 = d2.mergeSort();

// merge the two halves and return the result

// (d1 and d2 get garbage collected)

**return** d1.merge(d2);

}

I prefer the static version of merge because it treats the two decks as the same. However, both work well.

**Exercise 2/3:**

Because I could not find the Complex class, I wrote it myself. The changed abs() and equals() methods are included instead of the original ones given in the problem.

/\*\*

\* This class represents complex numbers

\* Written for homework

\* **@author** winst13

\*/

**public** **class** Complex

{

**private** **int** real;

**private** **int** imag;

/\*\*

\* Tests the methods in the Complex class

\* **@param** args

\*/

**public** **static** **void** main(String args[])

{

System.***out***.println((**new** Complex(3,4)).abs());//should print 5.0

System.***out***.println(*equals*(**new** Complex(3,4), **new** Complex(4,5)));//should print false

System.***out***.println(*equals*(**new** Complex(3,4), **new** Complex(3,4)));//should print true

}

/\*\*

\* Creates a new complex number

\* **@param** myreal

\* **@param** myimag

\*/

**public** Complex(**int** myreal, **int** myimag)

{

real = myreal;

imag = myimag;

}

/\*\*

\* Returns the absolute value, or magnitude, of the complex number

\* **@return**

\*/

**public** **double** abs()

{

**return** Math.*sqrt*(real \* real + imag \* imag);

}

/\*\*

\* Tests whether two complex numbers are equal

\* **@param** a

\* **@param** b

\* **@return**

\*/

**public** **static** **boolean** equals(Complex a, Complex b)

{

**return**(a.real == b.real && a.imag == b.imag);

}

}

**Exercise 4:**

1. The Rational class is shown below

**import** java.math.BigInteger;

/\*\*

\* This is the Rational class for Exercise 4 of Chapter 15

\* It contains all object methods

\* **@author** winst13

\*

\*/

**public** **class** Rational

{

**private** **int** numerator;

**private** **int** denominator;

/\*\*

\* Tests the class

\* **@param** args

\*/

**public** **static** **void** main(String args[])

{

Rational r1 = **new** Rational();

r1.setnumerator(9);

r1.setdenominator(12);

r1.printRational();

r1.negate();

r1.printRational();

r1.negate();

r1.invert();

r1.printRational();

System.***out***.println(r1.toDouble());

r1.reduce().printRational();

Rational r2 = **new** Rational(5,6);

r1.add(r2).printRational();

}

/\*\*

\* Constructor that creates a rational object with numerator 0 and denominator 1

\*/

**public** Rational()

{

numerator = 0;

denominator = 1;

}

**public** Rational(**int** mynumerator, **int** mydenominator)

{

numerator = mynumerator;

denominator = mydenominator;

}

**public** **void** printRational()

{

**if**(**this**.denominator==1)

System.***out***.println(**this**.numerator);

**else**

System.***out***.println(**this**.numerator+"/"+**this**.denominator);

}

**public** **void** negate()

{

numerator = -1\*numerator;

}

**public** **void** invert()

{

**int** newnumerator = denominator;

**int** newdenominator = numerator;

numerator = newnumerator;

denominator = newdenominator;

}

**public** **double** toDouble()

{

**double** decimalnum = (**double**) numerator;

**double** decimalden = (**double**) denominator;

**double** result = decimalnum/decimalden;

**return** result;

}

**public** Rational reduce()

{

BigInteger b1 = **new** BigInteger(""+**this**.numerator);

BigInteger b2 = **new** BigInteger(""+**this**.denominator);

BigInteger gcd = b1.gcd(b2);

**int** greatestcd = gcd.intValue();

Rational result = **new** Rational(**this**.numerator/greatestcd, **this**.denominator/greatestcd);

**return** result;

}

**public** Rational add(Rational r1)

{

**int** num = r1.numerator\***this**.denominator+**this**.numerator\*r1.denominator;

**int** den = r1.denominator\***this**.denominator;

Rational sum = **new** Rational(num,den);

**return** sum.reduce();

}

**public** **void** setnumerator(**int** newnumerator)

{

numerator = newnumerator;

}

**public** **void** setdenominator(**int** newdenominator)

{

denominator = newdenominator;

}

}

2.

Some mess-ups were made on purpose. Through these, I found that it was impossible to call an object method through the class. Likewise, though it was possible to call a static method through an object, it gave a warning.

3.

Usually, object methods are more concise. Usually, computations involving only one of the object in question are optimal for object methods, and computations involving two or more are optimal for class methods.

**Exercise 5:**

For this exercise, there are four classes. The first class, the CardSoln3 class, contains the main method, which tests the methods in the other classes.

/\*

\* Example code for Think Java (http://thinkapjava.com)

\*

\* Copyright(c) 2011 Allen B. Downey

\* GNU General Public License v3.0 (http://www.gnu.org/copyleft/gpl.html)

\*

\* @author Allen B. Downey

\* @author Winston Wang

\*/

**public** **class** CardSoln3 {

/\*

\* Test code.

\*/

**public** **static** **void** main(String[] args) {

Card card = **new** Card(1, 1);

card.print();

String s = card.toString();

System.***out***.println(s);

System.***out***.println(card);

Card card2 = **new** Card(1, 1);

System.***out***.println(card.equals(card2));

Deck deck = **new** Deck();

// check sortDeck

deck.shuffle();

deck.sort();

*checkSorted*(deck);

// check that findBisect finds each card

**int** index;

**for** (**int** i=0; i<52; i++) {

index = deck.findBisect(deck.cards[i], 0,

deck.cards.length-1);

**if** (index != i) {

System.***out***.println("Not found!");

}

}

// make a subdeck

Deck hand = deck.subdeck(8, 12);

hand.print();

// check that findBisect doesn't find a card that's not there

Card badCard = **new** Card(1, 1);

index = hand.findBisect(badCard, 0, hand.cards.length-1);

**if** (index != -1) {

System.***out***.println("Found?");

}

// check mergeSort

deck.shuffle();

deck = deck.mergeSort();

*checkSorted*(deck);

//This tests the deal() method, for part 4 of exercise 5

deck.shuffle();

PokerHand hand1 = deck.deal(5);

PokerHand hand2 = deck.deal(5);

PokerHand hand3 = deck.deal(5);

PokerHand hand4 = deck.deal(5);

System.***out***.println("");

hand1.print();

System.***out***.println("");

hand2.print();

System.***out***.println("");

hand3.print();

System.***out***.println("");

hand4.print();

System.***out***.println("");

//This test the poker hand methods, for parts 7 and 8

**int** n = 100000;

**int** numflush = 0;

**int** numtriple = 0;

**int** numpair = 0;

**int** numquad = 0;

**int** numtwopair = 0;

**int** numfullh = 0;

**int** numstraight = 0;

**int** numsflush = 0;

**for** (**int** i = 0; i < n; i++)

{

Deck newdeck = **new** Deck();

newdeck.shuffle();

PokerHand newhand = newdeck.deal(5);

**if** (newhand.hasFlush())

{

numflush++;

}

**if** (newhand.hasThreeKind())

{

numtriple++;

}

**if** (newhand.hasFourKind())

{

numquad++;

}

**if** (newhand.hasPair())

{

numpair++;

}

**if** (newhand.hasTwoPair())

{

numtwopair++;

}

**if** (newhand.hasFullHouse())

{

numfullh++;

}

**if** (newhand.hasStraight())

{

numstraight++;

}

**if** (newhand.hasStraightFlush())

{

numsflush++;

}

}

System.***out***.println("\n5 cards:");

System.***out***.println("Straight Flush:"+ ((**double**)numsflush)/((**double**)n)\*100);

System.***out***.println("Quadruple: "+ ((**double**)numquad)/((**double**)n)\*100);

System.***out***.println("Full Houlse: "+ ((**double**)numfullh)/((**double**)n)\*100);

System.***out***.println("Flush: "+ ((**double**)numflush)/((**double**)n)\*100);

System.***out***.println("Straight: "+ ((**double**)numstraight)/((**double**)n)\*100);

System.***out***.println("Triple: "+ ((**double**)numtriple)/((**double**)n)\*100);

System.***out***.println("Two Pair: "+ ((**double**)numtwopair)/((**double**)n)\*100);

System.***out***.println("Pair: "+ ((**double**)numpair)/((**double**)n)\*100);

//This test the poker hand methods, for parts 7 and 8

n = 100000;

numflush = 0;

numtriple = 0;

numpair = 0;

numquad = 0;

numtwopair = 0;

numfullh = 0;

numstraight = 0;

numsflush = 0;

**for** (**int** i = 0; i < n; i++)

{

Deck newdeck = **new** Deck();

newdeck.shuffle();

PokerHand newhand = newdeck.deal(7);

**if** (newhand.hasFlush())

{

numflush++;

}

**if** (newhand.hasThreeKind())

{

numtriple++;

}

**if** (newhand.hasFourKind())

{

numquad++;

}

**if** (newhand.hasPair())

{

numpair++;

}

**if** (newhand.hasTwoPair())

{

numtwopair++;

}

**if** (newhand.hasFullHouse())

{

numfullh++;

}

**if** (newhand.hasStraight())

{

numstraight++;

}

**if** (newhand.hasStraightFlush())

{

numsflush++;

}

}

System.***out***.println("\n7 cards:");

System.***out***.println("Straight Flush:"+ ((**double**)numsflush)/((**double**)n)\*100);

System.***out***.println("Quadruple: "+ ((**double**)numquad)/((**double**)n)\*100);

System.***out***.println("Full Houlse: "+ ((**double**)numfullh)/((**double**)n)\*100);

System.***out***.println("Flush: "+ ((**double**)numflush)/((**double**)n)\*100);

System.***out***.println("Straight: "+ ((**double**)numstraight)/((**double**)n)\*100);

System.***out***.println("Triple: "+ ((**double**)numtriple)/((**double**)n)\*100);

System.***out***.println("Two Pair: "+ ((**double**)numtwopair)/((**double**)n)\*100);

System.***out***.println("Pair: "+ ((**double**)numpair)/((**double**)n)\*100);

}

/\*

\* Checks that the deck is sorted.

\*/

**public** **static** **void** checkSorted(Deck deck) {

**for** (**int** i=0; i<51; i++) {

**int** flag = deck.cards[i].compareTo(deck.cards[i+1]);

**if** (flag != -1) {

System.***out***.println("Not sorted!");

}

}

}

}

The next class is the Deck class, which was provided (except for the deal() method, which I wrote)

/\*\*

\* A Deck contains an array of cards.

\*/

**public** **class** Deck {

Card[] cards;

/\*

\* Makes a Deck with room for n Cards (but no Cards yet).

\*/

**public** Deck(**int** n)

{

**this**.cards = **new** Card[n];

}

/\*

\* Makes an array of 52 cards.

\*/

**public** Deck()

{

**this**.cards = **new** Card [52];

**int** index = 0;

**for** (**int** suit = 0; suit <= 3; suit++)

{

**for** (**int** rank = 1; rank <= 13; rank++)

{

**this**.cards[index] = **new** Card(suit, rank);

index++;

}

}

}

/\*

\* Prints a deck of cards.

\*/

**public** **void** print() {

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

cards[i].print();

}

}

/\*

\* Returns index of the first location where card appears in deck.

\* Or -1 if it does not appear. Uses a linear search.

\*/

**public** **int** findCard (Card card) {

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

**if** (card.equals(cards[i])) {

**return** i;

}

}

**return** -1;

}

/\*

\* Returns index of a location where card appears in deck.

\* Or -1 if it does not appear. Uses a bisection search.

\*

\* Searches from low to high, including both.

\*

\* Precondition: the cards must be sorted from lowest to highest.

\*/

**public** **int** findBisect(Card card, **int** low, **int** high) {

**if** (high < low) **return** -1;

**int** mid = (high + low) / 2;

**int** comp = card.compareTo(cards[mid]);

**if** (comp == 0) {

**return** mid;

} **else** **if** (comp < 0) {

// card is less than cards[mid]; search the first half

**return** findBisect(card, low, mid-1);

} **else** {

// card is greater; search the second half

**return** findBisect(card, mid+1, high);

}

}

/\*

\* Chooses a random integer between low and high, including low,

\* not including high.

\*/

**public** **int** randInt(**int** low, **int** high) {

// Because Math.random can't return 1.0, and (int)

// always rounds down, this method can't return high.

**int** x = (**int**)(Math.*random*() \* (high-low) + low);

**return** x;

}

/\*

\* Swaps two cards in a deck of cards.

\*/

**public** **void** swapCards(**int** i, **int** j) {

Card temp = cards[i];

cards[i] = cards[j];

cards[j] = temp;

}

/\*

\* Shuffles the cards in a deck.

\*/

**public** **void** shuffle()

{

**for** (**int** i=0; i<cards.length; i++)

{

**int** j = randInt(i, cards.length-1);

swapCards(i, j);

}

}

/\*

\* Sorts a deck from low to high.

\*/

**public** **void** sort() {

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

**int** j = indexLowestCard(i, cards.length-1);

swapCards(i, j);

}

}

/\*

\* Finds the index of the lowest card between low and high,

\* including both.

\*/

**public** **int** indexLowestCard(**int** low, **int** high) {

**int** winner = low;

**for** (**int** i=low+1; i<=high; i++) {

**if** (cards[i].compareTo(cards[winner]) < 0) {

winner = i;

}

}

**return** winner;

}

/\*

\* Makes a new deck of cards with a subset of cards from the original.

\* The old deck and new deck share references to the same card objects.

\*/

**public** Deck subdeck(**int** low, **int** high) {

Deck sub = **new** Deck(high-low+1);

**for** (**int** i = 0; i<sub.cards.length; i++) {

sub.cards[i] = cards[low+i];

}

**return** sub;

}

/\*

\* Merges two sorted decks into a new sorted deck.

\*/

**public** **static** Deck merge(Deck d1, Deck d2) {

// create the new deck

Deck result = **new** Deck (d1.cards.length + d2.cards.length);

**int** choice; // records the winner (1 means d1, 2 means d2)

**int** i = 0; // traverses the first input deck

**int** j = 0; // traverses the second input deck

// k traverses the new (merged) deck

**for** (**int** k = 0; k < result.cards.length; k++) {

choice = 1;

// if d1 is empty, d2 wins; if d2 is empty, d1 wins; otherwise,

// compare the two cards

**if** (i == d1.cards.length)

choice = 2;

**else** **if** (j == d2.cards.length)

choice = 1;

**else** **if** (d1.cards[i].compareTo(d2.cards[j]) > 0)

choice = 2;

// make the new deck refer to the winner card

**if** (choice == 1) {

result.cards[k] = d1.cards[i]; i++;

} **else** {

result.cards[k] = d2.cards[j]; j++;

}

}

**return** result;

}

**public** Deck merge(Deck d2)

{

// create the new deck

Deck result = **new** Deck (**this**.cards.length + d2.cards.length);

**int** choice; // records the winner (1 means d1, 2 means d2)

**int** i = 0; // traverses the first input deck

**int** j = 0; // traverses the second input deck

// k traverses the new (merged) deck

**for** (**int** k = 0; k < result.cards.length; k++) {

choice = 1;

// if d1 is empty, d2 wins; if d2 is empty, d1 wins; otherwise,

// compare the two cards

**if** (i == **this**.cards.length)

choice = 2;

**else** **if** (j == d2.cards.length)

choice = 1;

**else** **if** (**this**.cards[i].compareTo(d2.cards[j]) > 0)

choice = 2;

// make the new deck refer to the winner card

**if** (choice == 1) {

result.cards[k] = **this**.cards[i]; i++;

} **else** {

result.cards[k] = d2.cards[j]; j++;

}

}

**return** result;

}

/\*

\* Sort the Deck using merge sort.

\*/

**public** Deck mergeSort() {

**if** (cards.length < 2) {

**return** **this**;

}

**int** mid = (cards.length-1) / 2;

// divide the deck roughly in half

Deck d1 = subdeck(0, mid);

Deck d2 = subdeck(mid+1, cards.length-1);

// sort the halves

d1 = d1.mergeSort();

d2 = d2.mergeSort();

// merge the two halves and return the result

// (d1 and d2 get garbage collected)

**return** d1.merge(d2);

}

/\*\*

\* This is for part 3 of exercise 5

\* It deals 5 cards, without replacement, to a PokerHand

\* **@return**

\*/

**public** PokerHand deal(**int** n)

{

PokerHand result = **new** PokerHand();

result.cards = **this**.subdeck(0, n-1).cards;

**this**.cards = **this**.subdeck(n,**this**.cards.length-1).cards;

**return** result;

}

}

The third class is the Card class, which I only added a getsuit() function to.

/\*

\* A Card represents a playing card with rank and suit.

\*/

**public** **class** Card {

**int** suit, rank;

**static** String[] *suits* = { "Clubs", "Diamonds", "Hearts", "Spades" };

**static** String[] *ranks* = { "narf", "Ace", "2", "3", "4", "5", "6",

"7", "8", "9", "10", "Jack", "Queen", "King" };

/\*

\* No-argument constructor.

\*/

**public** Card() {

**this**.suit = 0; **this**.rank = 0;

}

/\*

\* Constructor with arguments.

\*/

**public** Card(**int** suit, **int** rank) {

**this**.suit = suit; **this**.rank = rank;

}

/\*

\* Prints a card in human-readable form.

\*/

**public** **void** print() {

System.***out***.println(*ranks*[rank] + " of " + *suits*[suit]);

}

/\*

\* Prints a card in human-readable form.

\*/

**public** String toString() {

**return** *ranks*[rank] + " of " + *suits*[suit];

}

/\*

\* Return true if the cards are equivalent.

\*/

**public** **boolean** equals(Card that) {

**return** (**this**.suit == that.suit && **this**.rank == that.rank);

}

/\*

\* Compares two cards: returns 1 if the first card is greater

\* -1 if the seconds card is greater, and 0 if they are the equivalent.

\*/

**public** **int** compareTo(Card that) {

// first compare the suits

**if** (**this**.suit > that.suit) **return** 1;

**if** (**this**.suit < that.suit) **return** -1;

// if the suits are the same,

// use modulus arithmetic to rotate the ranks

// so that the Ace is greater than the King.

// WARNING: This only works with valid ranks!

**int** rank1 = (**this**.rank + 11) % 13;

**int** rank2 = (that.rank + 11) % 13;

// compare the rotated ranks

**if** (rank1 > rank2) **return** 1;

**if** (rank1 < rank2) **return** -1;

**return** 0;

}

**public** String getsuit()

{

**return** *suits*[suit];

}

}

The fourth and final class is the PokerHand class, which I wrote.

**import** java.util.Arrays;

/\*\*

\* For part 2 of exercise 5

\* This is the PokerHand class

\* **@author** winst13

\*

\*/

**class** PokerHand **extends** Deck

{

**public** PokerHand()

{

**super**(5);

}

**public** PokerHand(**int** num)

{

**super**(num);

}

/\*\*

\* This method returns whether a hand has a flush or not

\* **@return**

\*/

**public** **boolean** hasFlush()

{

String suit = cards[0].getsuit();

**this**.mergeSort();

**int** lengthofstreak = 0;

**for** (**int** i = 0; i < cards.length; i++)

{

**if** (suit == cards[i].getsuit())

{

lengthofstreak++;

}

**else**

{

suit = cards[i].getsuit();

lengthofstreak = 1;

}

**if** (lengthofstreak == 5)

{

**return** **true**;

}

}

**return** **false**;

}

/\*\*

\* This method returns whether a hand has a three of a kind

\* **@return**

\*/

**public** **boolean** hasThreeKind()

{

**if** (hasStreak(4))

{

**return** **false**;

}

**return** hasStreak(3);

}

/\*\*

\* This method returns whether a hand has a pair or not

\* It excludes the instances of two pairs

\* **@return**

\*/

**public** **boolean** hasPair()

{

**if** (hasStreak(2))

{

**int** rank = rankStreak(2);

**if** (hasStreak(2,rank))

{

**return** **false**;

}

}

**if** (hasStreak(4)||hasStreak(3))

{

**return** **false**;

}

**return** hasStreak(2);

}

/\*\*

\* This method returns whether a hand has a four of a kind

\* **@return**

\*/

**public** **boolean** hasFourKind()

{

**return** hasStreak(4);

}

/\*\*

\* This useful method is used when finding quads, trips, and pairs

\* **@param** n

\* **@return**

\*/

**public** **boolean** hasStreak(**int** n)

{

**int**[] ranks = **new** **int**[**this**.cards.length];

**for** (**int** i = 0; i < **this**.cards.length; i++)

{

ranks[i] = **this**.cards[i].rank;

}

Arrays.*sort*(ranks);

**int** lengthofstreak = 1;

**int** rank = -1;

**for** (**int** i = 0; i < **this**.cards.length; i++)

{

**if** (rank == ranks[i])

{

lengthofstreak++;

}

**else**

{

lengthofstreak = 1;

rank = ranks[i];

}

**if** (lengthofstreak == n)

{

**return** **true**;

}

}

**return** **false**;

}

/\*\*

\* This useful method is used when finding quads, trips, and pairs

\* It also allows the exception of one rank

\* **@param** n

\* **@return**

\*/

**public** **boolean** hasStreak(**int** n, **int** excluded)

{

**int**[] ranks = **new** **int**[**this**.cards.length];

**for** (**int** i = 0; i < **this**.cards.length; i++)

{

ranks[i] = **this**.cards[i].rank;

}

Arrays.*sort*(ranks);

**int** lengthofstreak = 1;

**int** rank = -1;

**for** (**int** i = 0; i < **this**.cards.length; i++)

{

**if** (rank == ranks[i]&&rank != excluded)

{

lengthofstreak++;

}

**else**

{

lengthofstreak = 1;

rank = ranks[i];

}

**if** (lengthofstreak == n)

{

**return** **true**;

}

}

**return** **false**;

}

/\*\*

\* Returns the rank of the streak of cards

\* **@return**

\*/

**public** **int** rankStreak(**int** n)

{

**int**[] ranks = **new** **int**[**this**.cards.length];

**for** (**int** i = 0; i < **this**.cards.length; i++)

{

ranks[i] = **this**.cards[i].rank;

}

Arrays.*sort*(ranks);

**int** lengthofstreak = 1;

**int** rank = -1;

**for** (**int** i = 0; i < **this**.cards.length; i++)

{

**if** (rank == ranks[i])

{

lengthofstreak++;

}

**else**

{

lengthofstreak = 1;

rank = ranks[i];

}

**if** (lengthofstreak == n)

{

**return** rank;

}

}

**return** -1;

}

/\*\*

\* Returns true if the hand has a two pair

\* **@return**

\*/

**public** **boolean** hasTwoPair()

{

**if** (hasStreak(2))

{

**int** rank = rankStreak(2);

**if** (hasStreak(2,rank))

{

**return** **true**;

}

**else**

{

**return** **false**;

}

}

**else**

{

**return** **false**;

}

}

/\*\*

\* This method returns whether a hand has a full house or not

\* **@return**

\*/

**public** **boolean** hasFullHouse()

{

**if** (hasStreak(3))

{

**int** rank = rankStreak(3);

**if** (hasStreak(2,rank))

{

**return** **true**;

}

**else**

{

**return** **false**;

}

}

**else**

{

**return** **false**;

}

}

/\*\*

\* This method returns whether the hand has a straight or not

\* **@return**

\*/

**public** **boolean** hasStraight()

{

**int**[] ranks = **new** **int**[**this**.cards.length];

**for** (**int** i = 0; i < **this**.cards.length; i++)

{

ranks[i] = **this**.cards[i].rank;

}

Arrays.*sort*(ranks);

**int** lengthofstreak = 1;

**int** rank = -1;

**for** (**int** i = 0; i < **this**.cards.length; i++)

{

**if** (rank == ranks[i]-1)

{

lengthofstreak++;

rank = ranks[i];

}

**else**

{

lengthofstreak = 1;

rank = ranks[i];

}

**if** (lengthofstreak == 5)

{

**return** **true**;

}

}

**return** **false**;

}

/\*\*

\* This method returns the highest rank in a 5-card straight

\* **@return**

\*/

**public** **int** rankStraight()

{

**int**[] ranks = **new** **int**[**this**.cards.length];

**for** (**int** i = 0; i < **this**.cards.length; i++)

{

ranks[i] = **this**.cards[i].rank;

}

Arrays.*sort*(ranks);

**int** lengthofstreak = 1;

**int** rank = -1;

**for** (**int** i = 0; i < **this**.cards.length; i++)

{

**if** (rank == ranks[i]-1)

{

lengthofstreak++;

rank = ranks[i];

}

**else**

{

lengthofstreak = 1;

rank = ranks[i];

}

**if** (lengthofstreak == 5)

{

**return** rank;

}

}

**return** -1;

}

/\*\*

\* This method returns the highest rank of a flush

\* **@return**

\*/

**public** **int** rankFlush()

{

String suit = cards[0].getsuit();

**this**.mergeSort();

**int** lengthofstreak = 0;

**for** (**int** i = 0; i < cards.length; i++)

{

**if** (suit == cards[i].getsuit())

{

lengthofstreak++;

}

**else**

{

suit = cards[i].getsuit();

lengthofstreak = 1;

}

**if** (lengthofstreak == 5)

{

**return** cards[i].rank;

}

}

**return** -1;

}

/\*\*

\* This method implements many helper functions. It can detect

\* whether a hand has a straight flush or not.

\* **@return**

\*/

**public** **boolean** hasStraightFlush()

{

**if** (hasFlush())

{

**int** endrank = rankFlush();

**if** (endrank==rankStraight())

{

**return** **true**;

}

**else**

{

**return** **false**;

}

}

**else**

{

**return** **false**;

}

}

}