Feb 07, 04 16:21 **problem1.txt** Page 1/1

```
# problem1.txt
# Dan R. K. Ports <drkp@mit.edu>
# 6.170 PS1-1, 2004/02/07

1. CLUBS, DIAMONDS, HEARTS, and SPADES are static so that one object
   corresponding to each suit is created as part of the class, rather
   than as part of each CardSuit object.

2. Using a separate class rather than integer constants makes it
   possible to distinguish a card suit from any other integer
   value. It also allows other methods to be added (for suit icons,
   etc).

3. this can not be null. If the constructor or some other method is
   being called, then it is referring to some object, and the this
   pointer is set.

4. The constructors are private so that no CardValue or CardSuit
   objects can be instantiated except for the static variables in the
   class.

5. The only CardValue and CardSuit objects are the static variables
   created in their respective classes, which are all different, so
   comparing the addresses (as the default Object.equals method does)
   is an acceptable test for equality.

6. Implementing the Comparable interface allows the Java Collections
   framework to understand the proper ordering of CardValue and
   CardSuit objects, and thus makes it easy to sort them.

7. The ordering of a Deck is significant (since decks of cards are
   shuffled, dealt from, etc). Thus, it is necessary to use an
   unsorted collection to represent one. A Hand does not have any
   particular ordering, and the cards are generally returned in sorted
   order, so it is reasonable to use a sorted collection.
```

so that only one instance of each suit exists in memory

2 3 4 3 3 4 14/20

Feb 07, 04 16:21 **problem5.txt** Page 1/1

```
Bankroll: 7900

Confirmation Code: 1694818
```

5/5

Overall: 98/100

```
package ps1.hand;

import ps1.playingcards.Card;
import ps1.playingcards.AcesLowComparator;

import java.util.Collections;
import java.util.Iterator;
import java.util.SortedSet;
import java.util.TreeSet;


/**
 *
 * The Hand class represents A single poker hand (set of playing
 * cards).  A typical poker hand usually contains 5 cards (and
 * contains no duplicates), but it might contain more or less cards,
 * depending on the poker game.  Each poker hand has a PokerRanking
 * (see PokerRanking class); note that a poker hand can sometimes be
 * associated with multiple poker rankings, in which case it is
 * associated with the highest ranked one.
 *
 * Hand objects are mutable; it possible to add and remove cards from
 * a Hand.
 *
 * @see ps1.playingcards.Card
 * @see PokerRanking
 *
 *** Implemented and passed all tests. -- drkp 2004/02/07
 */
public class Hand {

    // AF(c) = c represents the "poker hand" (set of playing cards)
    //         containing all of the cards in c.cards.

    // RI(c) = c.cards != null && c.cards contains no duplicates &&
    //         c.cards is sorted in increasing order (Ace is high)

    //
    // MEMBER VARIABLES
    //

    /**
     * The cards in this hand.
     */
    private SortedSet cards;

    //
    // METHODS
    //

    //-----------------------------------------
    /**
     * Creates a new hand containing no cards.
     *
     * @modifies this
     * @effects  creates a new <code>Hand</code> object with no initial cards
     */
    public Hand() {
        cards = new TreeSet();
    }

    //-----------------------------------------
    /**
     * Creates a new hand containing all of the cards in the Hand
     * object passed as an argument.  This is called a "copy
     * constructor", and is a common alternative to implementing
     * Object.clone.
     *
     *
```



```
     * @param hand  the hand being copied
     *
     * @modifies this
     * @effects  creates a new <code>Hand</code> object with the cards contained
 in hand.cards
     */
    public Hand(Hand hand) {
        cards = new TreeSet(hand.cards);
    }

    //-----------------------------------------
    /**
     * Adds the specified card to this hand.  If the specified card is null,
     * the method simply returns.
     *
     *
     * @param card  the card to be inserted
     *
     * @modifies this
     * @effects  <ul>
     *             <li>If <code>card</code> is null or is a duplicate of a card
already
     *                 in this deck, nothing is modified.</li>
     *             <li>Otherwise, adds <code>card</code> to this hand.</li>
     *           </ul>
     */
    public void addCard(Card card) {
        if (card != null)
            cards.add(card);
    }

    //-----------------------------------------
    /**
     * Removes the specified card from this hand.  If the specified card is null
     * or does not exist, the method simply returns.
     *
     * @param card  the card to be removed
     *
     *
     * @modifies this
     * @effects  <ul>
     *             <li>If <code>card</code> is null or does not exist in this ha
nd,
     *                 nothing is modified.</li>
     *             <li>Otherwise, removes <code>card</code> from this hand.</li>
     *           </ul>
     */
    public void removeCard(Card card) {
        if (card != null)
            cards.remove(card);
    }

    //-----------------------------------------
    /**
     * Tests whether the Card object passed as an argument is
     * contained in this poker hand.
     *
     * @param card  the Card that is tested
     *
     *
     * @effects  <ul>
     *             <li>If <code>card</code> is contained in this, then return tr
ue; otherwise,
     *             <li> return false
     *           </ul>
     */
    public boolean containsCard(Card card) {
        return cards.contains(card);
    }
```

```
    //-------------------------------------------
    /**
     *
     * @effects  Returns the number of cards currently in this hand.
     */
    public int getNumberOfCards() {
        return cards.size();
    }

    //-------------------------------------------
    /**
     * Lists the cards in this hand in the order in which they appear
     * (Aces are treated as greater than Kings). Consecutive calls to
     * this method should yield identical results.
     *
     *
     * @effects  Returns an <code>Iterator</code> of the cards in this hand, ord
ered
     *           from lowest card to highest.  Note that Aces are treated as gre
ater
     *           than Kings.
     */
    public Iterator listCardsAcesHigh() {

        //The unmodifable* methods in the Collections class protect
        //against Representation Exposure
        return Collections.unmodifiableCollection(cards).iterator();

    }

    //-------------------------------------------
    /**
     * Lists the cards in this hand in the order in which they usually
     * appear, except that Aces will be treated as lower than twos.
     * Consecutive calls to this method should yield identical results.
     *
     * @return an Iterator of the cards in this hand, in proper order
     *         (with the exception that Aces will appear lower than
     *         twos)
     *
     *
     *
     *
     * @effects  Returns an <code>Iterator</code> of the cards in this hand, ord
ered
     *           from lowest card to highest.  Note that Aces are treated as low
er
     *           than twos.
     */
    public Iterator listCardsAcesLow() {

        AcesLowComparator comparator = new AcesLowComparator();
        TreeSet acesLowSet = new TreeSet(comparator);

        acesLowSet.addAll(cards);

        return Collections.unmodifiableCollection(acesLowSet).iterator();
    }

    //-------------------------------------------
    /**
     * @effects returns a description of this hand's rank
     */
    public String describeHand() {
        PokerRanking rank = PokerRanking.rankHand(this);
        return rank.toString();
    }
```

nice!


```
    //-------------------------------------------
    /**
     * Compares this hand with the specified hand to determine which hand is
     * associated with a higher poker ranking.
     *
     * @param o  the Object to be compared
     * @return   a negative integer, zero, or a positive integer if this Hand
     *           is lower ranked, equally ranked, or higher ranked than the
     *           specified Hand, respectively
     * @exception ClassCastException if the specified object's type is not Hand
     * @exception NullPointerException if the specified object is null
     *
     *
     *
     *
     * @effects  This hand is ranked higher than hand <code>o</code> if the rank
ing
     *           associated with this hand is ranked higher than the ranking ass
ociated
     *           with hand <code>o</code>.  So:
     *           <ul>
     *             <li>If <code>o</code> is null, throws a <code>NullPointerExce
ption</code>.</li>
     *             <li>If <code>o</code> is not an instance of Hand, throws a
     *                 <code>ClassCastException</code>.</li>
     *             <li>If this hand is ranked higher than <code>o</code>, return
s any positive
     *                 integer.</li>
     *             <li>If this hand is ranked lower than <code>o</code>, returns
 any negative
     *                 integer.</li>
     *             <li>If this hand is equally ranked with <code>o</code>, retur
ns 0.</li>
     *           </ul>
     */
    public int compareTo(Object o) {
        PokerRanking otherHandsRank = PokerRanking.rankHand((Hand) o);
        PokerRanking rank = PokerRanking.rankHand(this);
        return rank.compareTo(otherHandsRank);
    }

    //-------------------------------------------
    /**
     * @effects  returns a description of this hand
     */
    public String toString() {
        Iterator remainingCards = listCardsAcesHigh();

        String description;
        if (!remainingCards.hasNext()) {
            description = "This hand contains no cards.";
        } else {
            description = "This hand contains the following cards: ";

            Card c = (Card) remainingCards.next();
            description += c.toString();

            while (remainingCards.hasNext()) {
                c = (Card) remainingCards.next();
                description += "," + c.toString();
            }
        }

        return description;
    }

    // this method is used to check representation invariants.  it will
    // become more important later in the term.
    public void checkRep() {
```

```
        if (cards == null)
            throw new RuntimeException("cards is null!");

        //test for duplicates
        Object[] cardsArray = cards.toArray();
        for (int i = 0; i < cardsArray.length; i++) {
            Card c = (Card) cardsArray[i];
            for (int j = i + 1; j < cardsArray.length; j++) {
                if (c == cardsArray[j])
                    throw new RuntimeException("Duplicate Card: " + c);
            }
        }

        //ensure sortedness
        // [no need to ensure sortedness because it is maintained by
        //  invariant in TreeSet]
    }

}
```



```
package ps1.playingcards;

import java.util.Comparator;


/**
 * AcesLowComparator is a comparator for card objects that overrides
 * the natural ordering, making aces the lowest value rather than the
 * highest.
 */

public class AcesLowComparator implements Comparator
{

    public int compare(Object o1, Object o2)
    {
        if (!(o1 instanceof Card && o2 instanceof Card))
            throw new ClassCastException();

        Card c1 = (Card) o1;
        Card c2 = (Card) o2;

        if (c1.getValue() == CardValue.ACE && c2.getValue() != CardValue.ACE)
            return -1;
        else if (c2.getValue() == CardValue.ACE && c1.getValue() != CardValue.AC
E)
            return 1;
        else
            return c1.compareTo(c2);
    }
}
```

why special cases?

```
package psl.playingcards;

/**
 * Card is a class representing single playing card consisting of a
 * value and a suit (e.g. [Ace,Spades],[10,Clubs]).  Cards are
 * immutable; once a Card has been created with a given value and
 * suit, that value and suit cannot be changed.
 *
 *** Implemented and passed all tests. --drkp 2004/02/07
 *
 */
public class Card implements Comparable {

    //  AF(c) = c represents a playing card with value of c.value and a
    //          suit of c.suit (e.g. Ace of Spades, 10 of Clubs)

    //  RI(c) = c.value != null && c.suit != null

    //
    // MEMBER VARIABLES
    //

    // the member variables are declared to be final because this class
    // is immutable.

    /**
     * The value of this card.
     */
    private final CardValue value;

    /**
     * The suit of this card.
     */
    private final CardSuit suit;

    //
    // METHODS
    //

    //-----------------------------------------
    /**
     * Creates a new playing card.
     *
     * @param aValue  the value of this card
     * @param aSuit   the suit of this card
     *
     * @modifies this
     * @effects  creates a new <code>Card</code> object
     */
    public Card(CardValue aValue, CardSuit aSuit) {
        value = aValue;
        suit = aSuit;
    }

    //-----------------------------------------
    /**
     * @effects  returns the <code>CardSuit</code> associated with this card
     */
    public CardSuit getSuit() {
        return suit;
    }

    //-----------------------------------------
    /**
     * @effects  returns the <code>CardValue</code> associated with this card
     */
    public CardValue getValue() {
        return value;
    }
```



```
    //-----------------------------------------
    /**
     * Compares this card with the specified card for order.  The
     * purpose of being able to compare cards is to be able to sort a
     * hand of cards.
     * <p>
     * Cards are ranked primarily by number, secondarily by suit.
     * That means that this card is ranked lower than another card if
     * one of these conditions is met:
     * <ul>
     * <li>This card's face value is less than the other card's face
     *     value; or
     * <li>Both cards' face values are equal, but this card's suit is
     *     ranked lower than the other card's suit.
     * </ul>
     *
     * EXAMPLE: [Ace,Clubs] is ranked higher than [10,Spades] because
     * face value is higher, but is ranked lower than [Ace,Spades]
     * because its suit is ranked lower.
     *
     * @param o  the Object to be compared
     * @exception ClassCastException if the specified object's type is
     *            not Card
     * @exception NullPointerException if the specified object is null
     *
     * @effects  <ul>
     *             <li>If <code>o</code> is not an instance of Card, throws a
     *                 <code>ClassCastException</code></li>
     *             <li>If <code>o</code> is null, throws a <code>NullPointerExce
ption</code></li>
     *             <li>Returns a negative integer, zero, or a positive integer i
f this
     *                 <code>Card</code> is less than, equal to, or greater than
 the
     *                 specified <code>Card</code>, respectively</li>
     *           </ul>
     */
    public int compareTo(Object o) {
        if (o == null) {
            throw new NullPointerException();
        }
        if (!(o instanceof Card)) {
            throw new ClassCastException();
        }
        if (value.compareTo( ((Card) o).value) == 0) {
            return suit.compareTo( ((Card) o).suit);
        } else {
            return value.compareTo( ((Card) o).value);
        }
    }

    //-----------------------------------------
    /**
     * Returns true if this card is equal to the other card.  Two
     * cards are equal if both their values and suits are identical.
     *
     * @param  otherCardObject  the other card
     *
     * @effects  returns true if both cards are equal; in all other cases,
     *           returns false
     */
    public boolean equals(Object otherCardObject) {

        if (otherCardObject == null || !(otherCardObject instanceof Card)) {
            return false;
        }
        return (this.compareTo(otherCardObject) == 0);
```

```
    }

    //-----------------------------------------
    /**
     * Returns a hashcode for this object.  This hashcode is the same
     * for all Cards equal to this one (as indicated by the equals
     * method).  Note that it is good practice to override the hashCode
     * method when redefining the equals method.
     *
     * @effects  returns a hashcode for this object; invoking this
     *           methods on two equal Cards results in the same hashcode
     */
    public int hashCode() {
        int suitMultiplier = CardSuit.SUITS.indexOf(suit);
        int valueInt = CardValue.VALUES.indexOf(value) + 1;
        return ((suitMultiplier * 13) + valueInt);
    }

    //-----------------------------------------
    /**
     * @effects returns a description of this card
     */
    public String toString() {
        return (value.toString() + " of " + suit.toString());
    }

    // this method is used to check representation invariants.  it will
    // become more important later in the term.
    public void checkRep() {
        if (value == null)
            throw new RuntimeException("card value is null!");
        if (suit == null)
            throw new RuntimeException("card suit is null!");
    }

}
```


25/25

```
package ps1.playingcards;

import java.util.Iterator;
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;

/**
 * The Deck class represents a standard Deck of playing cards.  When
 * constructed, the deck starts off holding the standard 52 playing
 * cards, one of each value-suit combination, with no duplicates.  It
 * is also possible to add and remove cards from the deck once it has
 * been created; therefore, the Deck data type is mutable.
 *
 *** Implemented and passed all tests. --drkp, 2004/02/07
 *
 * @see Card
 */
public class Deck {

    //
    // MEMBER VARIABLES
    //

    /**
     * The cards remaining in this deck.  The card at position "0" is
     * considered the card at the top of this deck.
     */
    List cards = new ArrayList();

    //
    // METHODS
    //

    //-----------------------------------------
    /**
     * Creates a new deck consisting of 52 Cards sorted first by suit
     * (Hearts, Spades, Diamonds, and finally Clubs), then by number
     * (starting with 2 and ending with an Ace).
     *
     *
     * @modifies this
     * @effects  creates a new <code>Deck</code> object
     */
    public Deck() {

        for (Iterator suitIter = CardSuit.SUITS.iterator(); suitIter.hasNext();
)
        {
            CardSuit suit = (CardSuit) suitIter.next();

            for (Iterator valIter = CardValue.VALUES.iterator(); valIter.hasNext
(); )
                cards.add(new Card((CardValue) valIter.next(), suit));
        }


        // PROBLEM SET HINT: look at /mit/6.170/www/psets/given/ps1/playingcards
/DeckTest.java
        //   for an idea of what this should do.
    }

    //-----------------------------------------
    /**
     * Adds the specified card to the bottom of this deck.  If the
     * specified card is null or a duplicate, the method simply
     * returns.
     *
     * @param c the Card to be inserted
```

```
     *
     *
     *
     * @modifies this
     * @effects  <ul>
     *             <li>If <code>c</code> is null or is a duplicate of a card alr
eady
     *                 in this deck, nothing is modified.</li>
     *             <li>Otherwise, adds <code>c</code> to the bottom of this deck
.</li>
     *           </ul>
     */
    public void addCard(Card c) {
        if (c != null && !(cards.contains(c)))
            cards.add(c);
    }

    //-----------------------------------------
    /**
     * Removes the specified card from this deck.  If the specified
     * card is null or does not exist in this deck, the method simply
     * returns.
     *
     * @param c the card to be removed
     *
     *
     *
     * @modifies this
     * @effects  <ul>
     *             <li>If <code>c</code> is null or does not exist in this deck,
     *                 nothing is modified.</li>
     *             <li>Otherwise, removes <code>c</code> from this deck.</li>
     *           </ul>
     */
    public void removeCard(Card c) {
        if (c != null && cards.contains(c))
            cards.remove(cards.indexOf(c));
    }

    //-----------------------------------------
    /**
     * Randomly permutates the order of the cards in this deck.
     * Consecutive calls to {@link #listCards listCards} after
     * shuffling inbetween should result in a different ordering of
     * the cards.
     *
     *
     *
     * @modifies this
     * @effects  randomly rearranges the cards in this deck
     */
    public void shuffle() {
        Collections.shuffle(cards);
    }

    //-----------------------------------------
    /**
     * Removes n cards from the top of the deck and returns them.
     *
     * @param n  the number of cards to "deal" from this deck
     *
     *
     *
     * @modifies this
     * @effects  <ul>
     *             <li>If <code>n</code> is less than or equal to 0, returns an
     *                 <code>Iterator</code> over 0 elements (i.e. an empty
     *                 Iterator)</li>
     *             <li>If <code>n</code> is greater than 0 but less than the num
```



```
ber of
     *                 cards currently in this deck, returns an <code>Iterator</
code>
     *                 containing the top n cards in this deck; also, the cards
     *                 that appear in this Iterator will no longer appear in thi
s deck</li>
     *             <li>If <code>n</code> is greater than 0 and greater than the
number of
     *                 cards currently in this deck, returns an <code>Iterator</
code>
     *                 containing all the cards in this deck; also, the cards
     *                 that appear in this Iterator will no longer appear in thi
s deck</li>
     *           </ul>
     */
    public Iterator dealCards(int n) {
        if (n <= 0) {
            return ((new ArrayList()).iterator());
        }

        // Figure out how many cards we will be dealing.
        int numCardsToRemove = n;
        if (cards.size() <= n) {
            numCardsToRemove = cards.size();
        }

        // Now deal the cards.
        List dealtCards = cards.subList(0, numCardsToRemove);
        List finalListOfCards = new ArrayList(dealtCards);

        // Remove the dealt cards from the deck.
        dealtCards.clear();

        return finalListOfCards.iterator();
    }

    //-----------------------------------------
    /**
     * Returns an iterator that will lists the cards in this deck, from top to b
ottom.
     *
     * @effects  returns an <code>Iterator</code> of the cards in this deck, ord
ered
     *           from the top of this deck to its bottom
     */
    public Iterator listCards() {
        List unmodifiableListOfCards = Collections.unmodifiableList(cards);
        return unmodifiableListOfCards.iterator();
    }

    //-----------------------------------------
    /**
     * Returns a description of this deck.
     *
     *
     * @effects  returns a description of this deck
     */
    public String toString() {
        Iterator remainingCards = listCards();

        String description;
        if (!remainingCards.hasNext()) {
            description = "This deck contains no cards.";
        } else {
            description = "This deck contains the following cards: ";

            Card c = (Card) remainingCards.next();
            description += c.toString();
```

```
            while (remainingCards.hasNext()) {
                c = (Card) remainingCards.next();
                description += "," + c.toString();
            }
        }

        return description;
    }
}
```