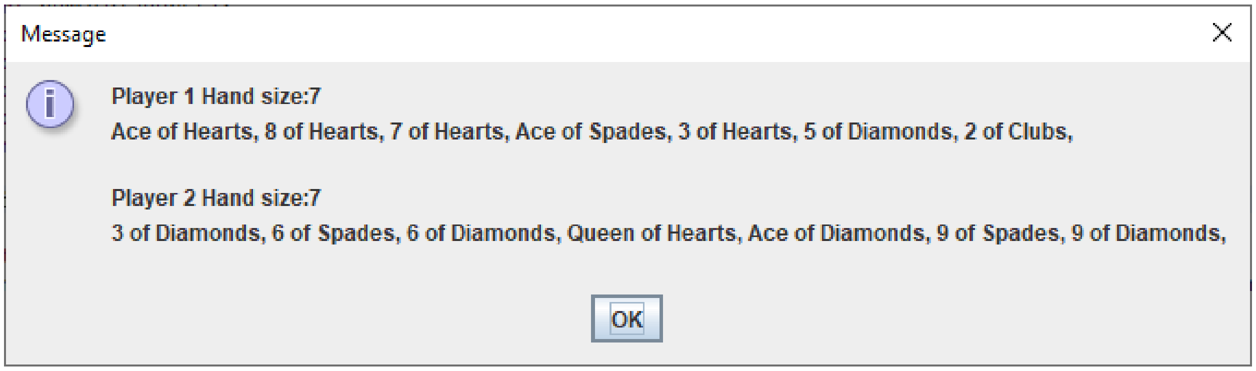
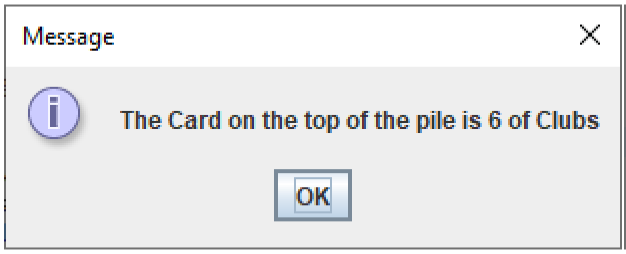
Noah Streveler

Assignment 4

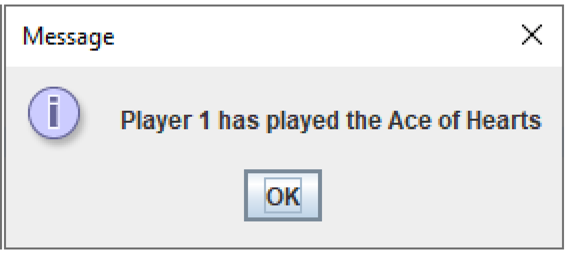

Message

Player 1 Hand size:7
Ace of Hearts, 8 of Hearts, 7 of Hearts, Ace of Spades, 3 of Hearts, 5 of Diamonds, 2 of Clubs,

Player 2 Hand size:7
3 of Diamonds, 6 of Spades, 6 of Diamonds, Queen of Hearts, Ace of Diamonds, 9 of Spades, 9 of Diamonds,


Message

The Card on the top of the pile is 6 of Clubs


Message

Player 1 has played the Ace of Hearts

```
/**
 *  An object of type Deck represents a deck of playing cards.  The deck
 *  is a regular poker deck that contains 52 regular cards and that can
 *  also optionally include two Jokers.
 */
public class Deck {

    /**
     * An array of 52 or 54 cards. A 54-card deck contains two Jokers,
     * in addition to the 52 cards of a regular poker deck.
     */
    private Card[] deck;
    private int currentSize = 0;
    private int cardCt = 0;

    /**
     * Keeps track of the number of cards that have been dealt from
     * the deck so far.
     */
    private int cardsUsed;

    /**
     * Constructs a regular 52-card poker deck. Initially, the cards
     * are in a sorted order.  The shuffle() method can be called to
     * randomize the order.  (Note that "new Deck()" is equivalent
     * to "new Deck(false)".)
     */
    public Deck() {
        this(false);  // Just call the other constructor in this class.
}

/**
 * Constructs a poker deck of playing cards, The deck contains
 * the usual 52 cards and can optionally contain two Jokers
 * in addition, for a total of 54 cards.   Initially the cards
 * are in a sorted order.  The shuffle() method can be called to
 * randomize the order.
 * @param includeJokers if true, two Jokers are included in the deck; if false,
 * there are no Jokers in the deck.
 */
public Deck(boolean includeJokers) {
    if (includeJokers){
        deck = new Card[54];
        currentSize = 54;
    }
    else{
        deck = new Card[52];
        currentSize = 52;
    }
    for ( int suit = 0; suit <= 3; suit++ ) {
        for ( int value = 1; value <= 13; value++ ) {
            deck[cardCt] = new Card(value,suit);
            cardCt++;
        }
    }
    if (includeJokers) {
        deck[52] = new Card(0, Card.JOKER);
        deck[53] = new Card(14, Card.JOKER);
    }
    cardsUsed = 0;
}

/**
 * Put all the used cards back into the deck (if any), and
 * shuffle the deck into a random order. This is one of the ways
 * shuffle can be done.
 */
public void shuffle() {
    for ( int i = deck.length-1; i > 0; i-- ) {
        int rand = (int)(Math.random()*(i+1));
        Card temp = deck[i];
        deck[i] = deck[rand];
        deck[rand] = temp;
    }
    cardsUsed = 0;
}

/**
 * As cards are dealt from the deck, the number of cards left
 * decreases.  This function returns the number of cards that
 * are still left in the deck.  The return value would be
 * 52 or 54 (depending on whether the deck includes Jokers)
 * when the deck is first created or after the deck has been
 * shuffled.  It decreases by 1 each time the dealCard() method
```

```java
     * is called.
     */
    public int cardsLeft() {
        return currentSize - cardsUsed;
    }

    /**
     * Removes the next card from the deck and return it.  It is illegal
     * to call this method if there are no more cards in the deck.  You can
     * check the number of cards remaining by calling the cardsLeft() function.
     * @return the card which is removed from the deck.
     * @throws IllegalStateException if there are no cards left in the deck
     */
    public Card dealCard() {
        if (cardsUsed < currentSize){
             Card toDeal = deck[cardsUsed];
             cardsUsed++;
             return toDeal;
        }
        else return null;
        // Programming note:  Cards are not literally removed from the array
        // that represents the deck.  We just keep track of how many cards
        // have been used.
    }

    /**
     * Test whether the deck contains Jokers.
     * @return true, if this is a 54-card deck containing two jokers, or false if
     * this is a 52 card deck that contains no jokers.
     */
    public boolean hasJokers() {
        return (deck.length == 54);
    }

    public void removeCard(int value, int suit){
       for(int i = 0; i < deck.length; i++){
             if(deck[i].getValue() == value && deck[i].getSuit() == suit){
                    deck[i] = deck[deck.length-1];
                    currentSize--;
                    break;
             }
       }
    }
}


import java.util.ArrayList;
import java.util.List;

public abstract class GameControl {
       protected IGameView view = new IOHandler();
       protected Deck deck;
       protected List<Player> players;
       protected Hand[] group;
       protected Hand discardPile;
```

```
	protected Hand extrasPile;
	protected int num;

	public GameControl(){
		deck = new Deck();
		deck.shuffle();
		players = new ArrayList<Player>();
		init();
		discardPile = new Hand();
		extrasPile = new Hand();
	}
	public void runGame(){
		int numRounds = 1;
		Character input = view.getInput("Play? (t/f)" + "?");
		if( input != 't') return;
	    do {
		num = numOfPlayers();
		group = new Hand[num];
		    for(int i = 0; i < group.length; i++) {
			group[i] = new Hand(); //Instantiate
			}
			startGame();
		do{
			view.display("Round " + numRounds + ":");
			view.display("The Card on the top of the pile is " +
discardPile.getCard(discardPile.getCardCount()-1));
			view.display(playersHands());
			playRound();
			numRounds++;
		}while(!isEmpty() && isWinner() == -1);
		endGame();
	    } while ( ((char) view.getInput("Play again (t/f)" + "?")) == 't');
	}
	abstract void init();
	abstract int numOfPlayers();
	abstract void startGame();
	abstract void playRound();
	abstract void endGame();
	abstract boolean isEmpty();
	abstract int isWinner();

	public String playersHands(){
		String hands = "";
		for(int i = 0; i < num; i++){
			hands += "Player " + (i + 1) + " Hand size:" +
group[i].getCardCount() + "\n" + group[i].displayHand(group[i]) + "\n\n";
		}
		return hands;
	}
}


interface IGameView{
	void getResult(String prompt);
	void display(String message);
```

```
      <T> T getInput(String msg);
}
```

```
import java.util.ArrayList;
import java.util.Collections;

/**
 * An object of type Hand represents a hand of cards.  The
 * cards belong to the class Card.  A hand is empty when it
 * is created, and any number of cards can be added to it.
 */
public class Hand {
    private ArrayList<Card> hand;   // The cards in the hand.

    public Hand() {
        hand = new ArrayList<Card>();
    }

    public void clear() {
        hand.clear();
    }

    /**
     * Add a card to the hand.  It is added at the end of the current hand.
     * @param c the non-null card to be added.
     * @throws NullPointerException if the parameter c is null.
     */
    public void addCard(Card c) {
        if (c == null)
            throw new NullPointerException("Can't add a null card to a hand.");
        hand.add(c);
    }

    /**
     * Remove a card from the hand, if present.
     * @param c the card to be removed.  If c is null or if the card is not in
     * the hand, then nothing is done.
     */
    public void removeCard(Card c) {
        hand.remove(c);
    }

    /**
     * Remove the card in a specified position from the hand.
     * @param position the position of the card that is to be removed, where
     * positions are starting from zero.
     * @throws IllegalArgumentException if the position does not exist in
     * the hand, that is if the position is less than 0 or greater than
     * or equal to the number of cards in the hand.
     */
    public void removeCard(int position) {
        if (position < 0 || position >= hand.size())
            throw new IllegalArgumentException("Position does not exist in hand: "
                    + position);
        hand.remove(position);
    }

    /**
     * Returns the number of cards in the hand.
     */
    public int getCardCount() {
        return hand.size();
    }

    /**
     * Gets the card in a specified position in the hand.  (Note that this card
     * is not removed from the hand!)
     * @param position the position of the card that is to be returned
     * @throws IllegalArgumentException if position does not exist in the hand
     */
    public Card getCard(int position) {
        if (position < 0 || position >= hand.size())
            throw new IllegalArgumentException("Position does not exist in hand: "
                    + position);
        return hand.get(position);
    }

    /**
     * Sorts the cards in the hand so that cards of the same suit are
     * grouped together, and within a suit the cards are sorted by value.
     * Note that aces are considered to have the lowest value, 1. --- sorting is
similar to "selection sort"
     */
    public void sortBySuit() {
        ArrayList<Card> newHand = new ArrayList<Card>();
        while (hand.size() > 0) {
            int pos = 0;  // Position of minimal card.
            Card c = hand.get(0);  // Minimal card.
            for (int i = 1; i < hand.size(); i++) {
                Card c1 = hand.get(i);
                if ( c1.getSuit() < c.getSuit() ||
                        (c1.getSuit() == c.getSuit() && c1.getValue() < c.getValue())
) {
                    pos = i;
                    c = c1;
                }
            }
            hand.remove(pos);
            newHand.add(c);
        }
        hand = newHand;
    }

    /**
     * Sorts the cards in the hand so that cards of the same value are
     * grouped together.  Cards with the same value are sorted by suit.
     * Note that aces are considered to have the lowest value, 1.
     */
    public void sortByValue() {
        ArrayList<Card> newHand = new ArrayList<Card>();
        while (hand.size() > 0) {
            int pos = 0;  // Position of minimal card.
            Card c = hand.get(0);  // Minimal card.
            for (int i = 1; i < hand.size(); i++) {
                Card c1 = hand.get(i);
                if ( c1.getValue() < c.getValue() ||
                        (c1.getValue() == c.getValue() && c1.getSuit() < c.getSuit())
) {
                    pos = i;
                    c = c1;
                }
            }
            hand.remove(pos);
            newHand.add(c);
        }
        hand = newHand;
    }

    public String displayHand(Hand group) {
             String str = "";
       for(int i = 0; i < group.getCardCount(); i++){
                    str += group.getCard(i).toString() + ", ";
             }
       return str;
    }

       public boolean isHandEmpty() {
             return hand.size() == 0;
       }
}//end of class Hand
```

```
import java.util.*;
import java.time.*;
import javax.swing.JOptionPane;

public class OldMaid {
   public static void main(String[] args) {
          GameControl controller= new GameController();
          controller.runGame();
   }
}

class IOHandler implements IGameView{
       //Scanner  sc = new Scanner(System.in);
       char input;
       private static char[] matches = new char[]{'f', 't'};

       @Override
       public void display(String message) {
              JOptionPane.showMessageDialog(null, message);
       }

       @Override
       public Character getInput(String msg) {
              boolean isCorrectInput = false;
              do {
             input = JOptionPane.showInputDialog(msg).charAt(0);
             input = Character.toLowerCase(input);
             for(int i = 0; i < matches.length; i++){
              if (input == matches[i]) {
                     return new Character(input);
              }
             }
             System.out.print("Please respond with an expected character:  ");
         } while (!isCorrectInput);
              return null;
       }

       @Override
       public void getResult(String prompt) {
              // TODO Auto-generated method stub
       }
}

class GameController extends GameControl{  //game model + game control
       Hand prior = null;
       Hand current = null;
       int currPlay = 0;
       public GameController(){
              super();
```

```
	}

	@Override
	public void init(){
		view.display("This program lets you play a card game: Crazy Eights\n");
		//players.add(new OMHumanPlayer("1"));
	    //for(int i = 1; i <= 3; i++){
		//players.add(new OldMaidPlayer(""+ (i+1)));
	    //}
	}

	@Override
	public void startGame(){
		dealCards();
	}

	/**
	  * Play one round of Old Maid --  each player has played once
	 */
	 @Override
	 public void playRound() {
		 for(int i = 0; i < group.length; i++){
			 current = group[i];
			 currPlay = i + 1;
			 playTurn(i);
			 /*
			 if(current.isHandEmpty()) continue;
			 prior = getPriorPlayer(i);
			 if(prior == null) return;
			 if(current instanceof OMHumanPlayer){
				 current.play();
				 continue;
			 }
			 Card c = prior.giveCard();
			 current.play(c);
			 */
		 }
	 }

	 @Override
	 public void endGame() {
		 if(isWinner() != -1) {
			   view.display("Congratulations to Player " + isWinner() + " for 
Winning!");
		    }
	 }

	 @Override
	 public int numOfPlayers() {
		 int player = Integer.parseInt(JOptionPane.showInputDialog("How many 
people will be playing (2-6)?"));
		 if(player < 2 && player > 6) {
			 System.out.println("I'm sorry, but the option that you chose is 
not available, please try again");
			 System.exit(0);
```

```
            }
            return player;
        }

        @Override
        public boolean isEmpty(){
            if(extrasPile.isHandEmpty()) {
                while( 1 < discardPile.getCardCount()) {
                    extrasPile.addCard(discardPile.getCard(1));
                    discardPile.removeCard(1);

                }
                view.display("The deck was restocked");
            }
            return false;
        }

        @Override
        public int isWinner(){
            for(int i = 0; i < group.length; i++){
                if (group[i].isHandEmpty()){
                    return (i + 1);
                }
            }
            return -1;
        }

        private void grabNextCard(int num) {
            group[num].addCard(extrasPile.getCard(0));
            view.display("Player " + (currPlay) + " had to draw and drew the " +
extrasPile.getCard(0));
            extrasPile.removeCard(0);
        }

        private boolean hasMatch() {
                return findMatch() != -1;//If -1 there is no match
        }

       private int findMatch() {
            int holdEight = -1;
            boolean match = false;
            Card p;
            Card c = null;
            for(int i = 0; i < current.getCardCount(); i++) {
                p = discardPile.getCard(discardPile.getCardCount() - 1);
                c = current.getCard(i);

                if(c.getValue() == 8) holdEight = i;//Finds an 8, but doesn't use
it right away
                if(p.getValue() == c.getValue()) {//Looking for other rank
matches so doesn't quit right away
                    discardPile.addCard(c);
                    view.display("Player " + (currPlay) + " has played the " +
c);
                    current.removeCard(i);
```

```
                        i--;
                        match = true;
                    }
                    if(p.getSuit() == c.getSuit() && !match) {//Returns that it found
a match
                        discardPile.addCard(c);
                        view.display("Player " + (currPlay) + " has played the " +
c);
                        current.removeCard(i);
                        return 0;
                    }
                }
                if(match) return 0;//If it found a match returns
                if(holdEight > -1) {//If it has to play an Eight it will
                    discardPile.addCard(current.getCard(holdEight));
                    view.display("Player " + (currPlay) + " has played the " + c);
                    current.removeCard(holdEight);
                    return 0;
                }
                return -1;
            }

             private void playTurn(int i) {
                while(!hasMatch() && !isEmpty()) {
                    grabNextCard(i);
                }
             }

            private void dealCards(){
                for(int i = 0; i < num * 7; i++) {//Giving the players their Cards
                  for(int j = 0; i < num * 7; j++) {
                        Card newCard = deck.dealCard();
                        group[j].addCard(newCard);

                        if(j == (group.length - 1))
                            j=-1;
                        i++;
                  }
                }

                Card newCard = deck.dealCard();
                discardPile.addCard(newCard);//Getting the starting pile Card

                for(int i = 0; i < (52 - (num * 7)) - 1; i++) {
                    newCard = deck.dealCard();
                    extrasPile.addCard(newCard);//Giving the grabbable Cards
                }

                while(discardPile.getCard(0).getValue() == 8) {//Incase the starting
card is an 8
                    Card temp = discardPile.getCard(0);
                    discardPile.addCard(temp);
                    discardPile.removeCard(0);
                }
             }
```

```
}
```

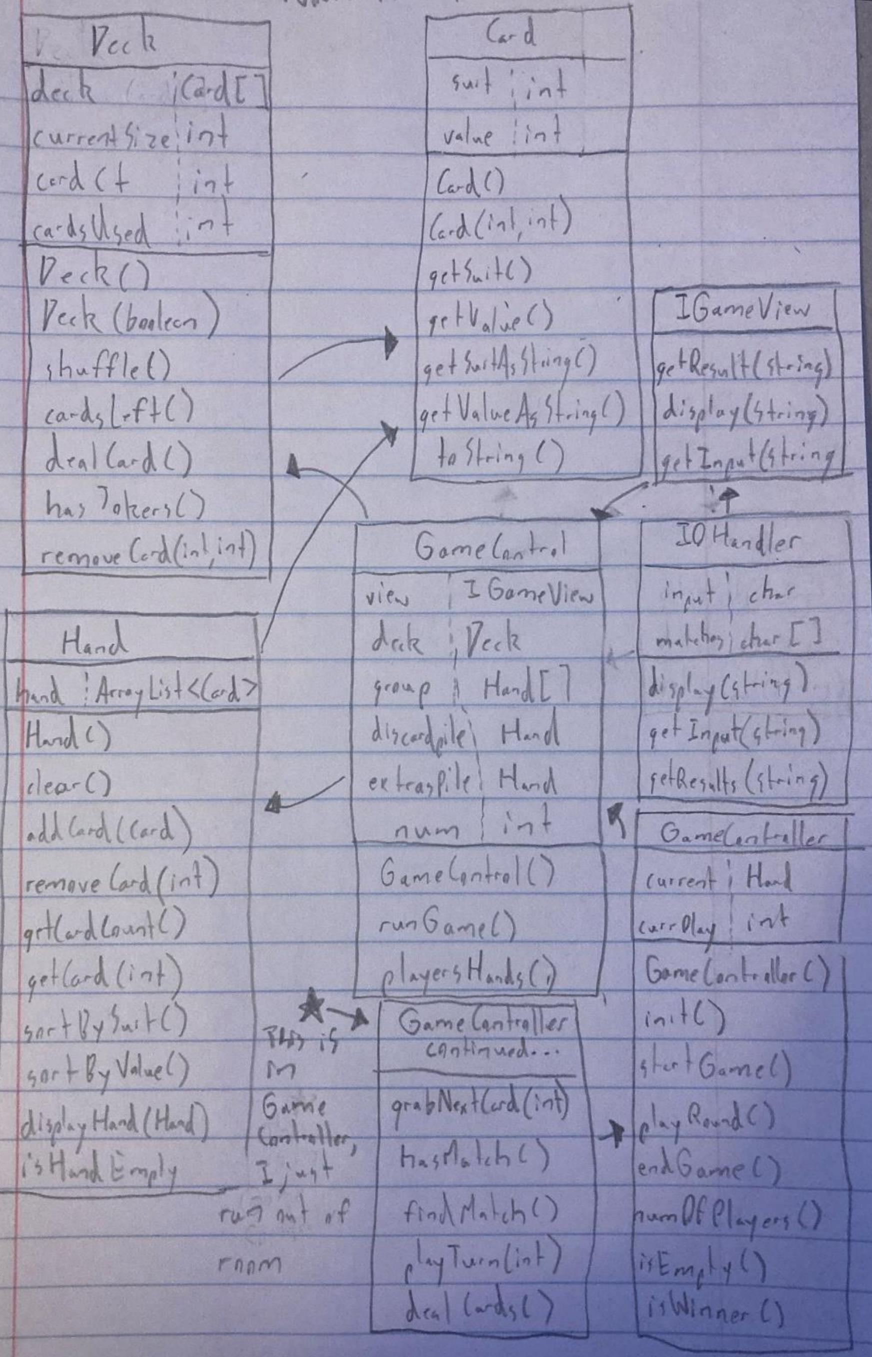
```
public class Card {

   public final static int SPADES = 0;   // Codes for the 4 suits, plus Joker.
   public final static int HEARTS = 1;
   public final static int DIAMONDS = 2;
   public final static int CLUBS = 3;
   public final static int JOKER = 4;

   public final static int ACE = 1;      // Codes for the non-numeric cards.
   public final static int JACK = 11;    //   Cards 2 through 10 have their
   public final static int QUEEN = 12;   //   numerical values for their codes.
   public final static int KING = 13;

   /**
    * This card's suit, one of the constants SPADES, HEARTS, DIAMONDS,
    * CLUBS, or JOKER.  The suit cannot be changed after the card is
    * constructed.
    */
   private final int suit;

   /**
    * The card's value.  For a normal card, this is one of the values
    * 1 through 13, with 1 representing ACE.  For a JOKER, the value
    * can be anything.  The value cannot be changed after the card
    * is constructed.
    */
   private final int value;

   /**
    * Creates a Joker, with 1 as the associated value.  (Note that
    * "new Card()" is equivalent to "new Card(0, Card.JOKER)".)
    */
   public Card() {
      suit = JOKER;
      value = 0;
   }

   /**
    * Creates a card with a specified suit and value.
```

```
     * @param theValue the value of the new card.  For a regular card (non-joker),
     * the value must be in the range 1 through 13, with 1 representing an Ace.
     * You can use the constants Card.ACE, Card.JACK, Card.QUEEN, and Card.KING.
     * For a Joker, the value can be anything.
     * @param theSuit the suit of the new card.  This must be one of the values
     * Card.SPADES, Card.HEARTS, Card.DIAMONDS, Card.CLUBS, or Card.JOKER.
     * @throws IllegalArgumentException if the parameter values are not in the
     * permissible ranges
     */
    public Card(int v, int s) {
       if (s != SPADES && s!= HEARTS && s != DIAMONDS &&
             s != CLUBS && s != JOKER)
          throw new IllegalArgumentException("Illegal card suit");
       if (s != JOKER && (v < 1 || v > 13))
          throw new IllegalArgumentException("Illegal card value");
       value = v;
       suit = s;
    }

    public int getSuit() {
       return suit;
    }

    public int getValue() {
       return value;
    }

    /**
     * Returns a String representation of the card's suit.
     * @return one of the strings "Spades", "Hearts", "Diamonds", "Clubs"
     * or "Joker".
     */
    public String getSuitAsString() {
       switch ( suit ) {
       case SPADES:   return "Spades";
       case HEARTS:   return "Hearts";
       case DIAMONDS: return "Diamonds";
       case CLUBS:    return "Clubs";
       default:       return "Joker";
       }
    }

    /**
     * Returns a String representation of the card's value.
     * @return for a regular card, one of the strings "Ace", "2",
     * "3", ..., "10", "Jack", "Queen", or "King".  For a Joker, the
     * string is always numerical.
     */
    public String getValueAsString() {
       if (suit == JOKER)
          return "" + value;
       else {
          switch ( value ) {
          case 1:   return "Ace";
          case 2:   return "2";
```

```
         case 3:   return "3";
         case 4:   return "4";
         case 5:   return "5";
         case 6:   return "6";
         case 7:   return "7";
         case 8:   return "8";
         case 9:   return "9";
         case 10:  return "10";
         case 11:  return "Jack";
         case 12:  return "Queen";
         default:  return "King";
         }
      }
   }

   /**
    * Returns a string representation of this card, including both
    * its suit and its value (except that for a Joker with value 1,
    * the return value is just "Joker").  Sample return values
    * are: "Queen of Hearts", "10 of Diamonds", "Ace of Spades",
    * "Joker", "Joker #2"
    */
   public String toString() {
      if (suit == JOKER) {
         if (value == 0)
            return "Joker";
         else
            return "Joker #" + value;
      }
      else
         return getValueAsString() + " of " + getSuitAsString();
   }
} // end class Card
```

UML

Noah Streveler


Deck
deck : Card[]
currentSize : int
cardCt : int
cardsUsed : int
Deck()
Deck(boolean)
shuffle()
cardsLeft()
dealCard()
hasJokers()
removeCard(int,int)
Card
suit : int
value : int
Card()
Card(int,int)
getSuit()
getValue()
getSuitAsString()
getValueAsString()
toString()
IGameView
getResult(string)
display(string)
getInput(string
Hand
hand : ArrayList<Card>
Hand()
clear()
addCard(Card)
removeCard(int)
getCardCount()
getCard(int)
sortBySuit()
sortByValue()
displayHand(Hand)
isHandEmpty
GameControl
view : IGameView
deck : Deck
group : Hand[]
discardpile : Hand
extraspile : Hand
num : int
GameControl()
runGame()
playersHands()
IOHandler
input : char
matches : char[]
display(string)
getInput(string)
getResults(string)
GameController
current : Hand
currPlay : int
GameController()
init()
startGame()
playRound()
endGame()
numOfPlayers()
isEmpty()
isWinner()
★
This is in GameController, I just run out of room
GameController continued...
grabNextCard(int)
hasMatch()
findMatch()
playTurn(int)
dealCards()