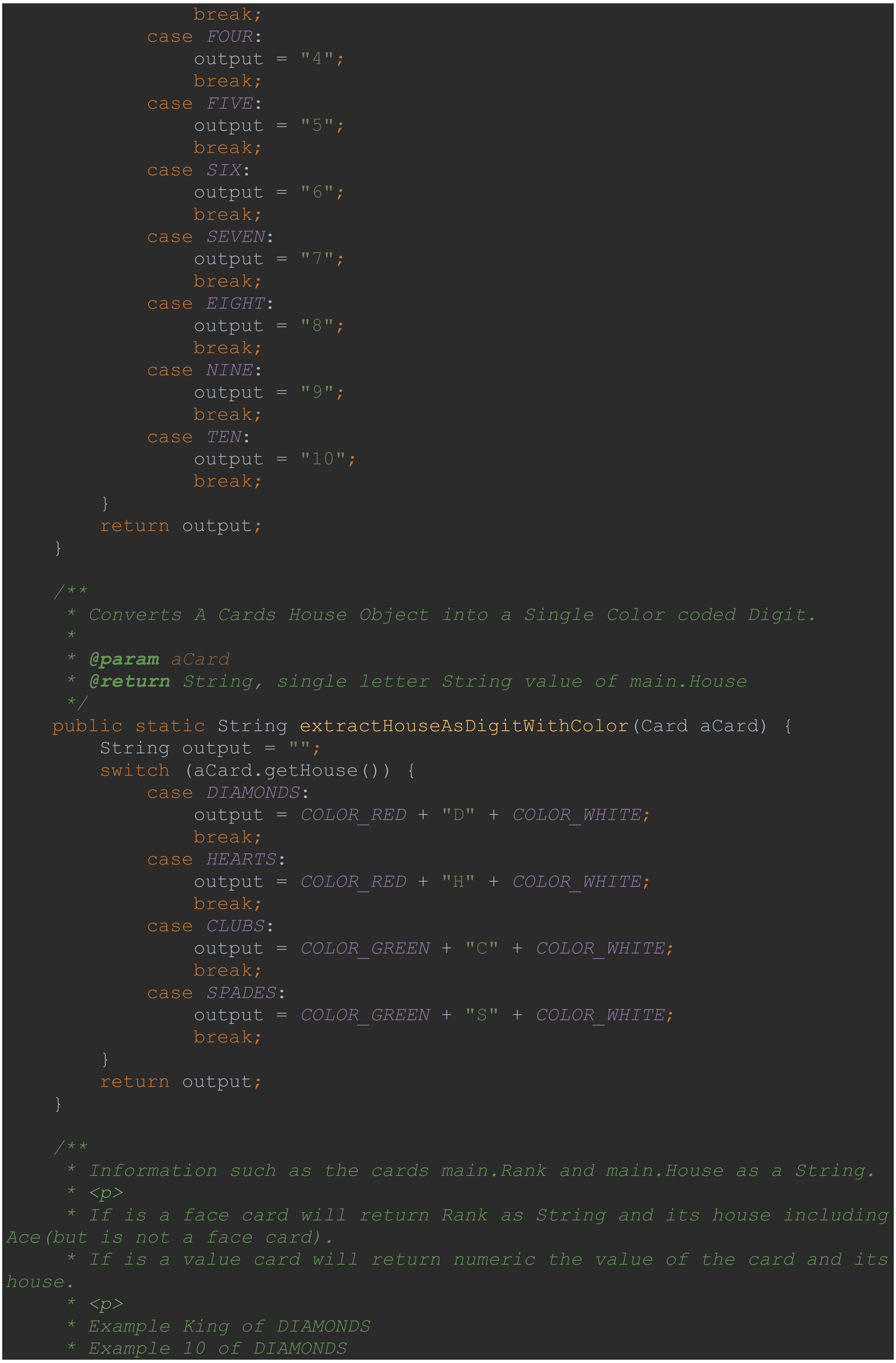
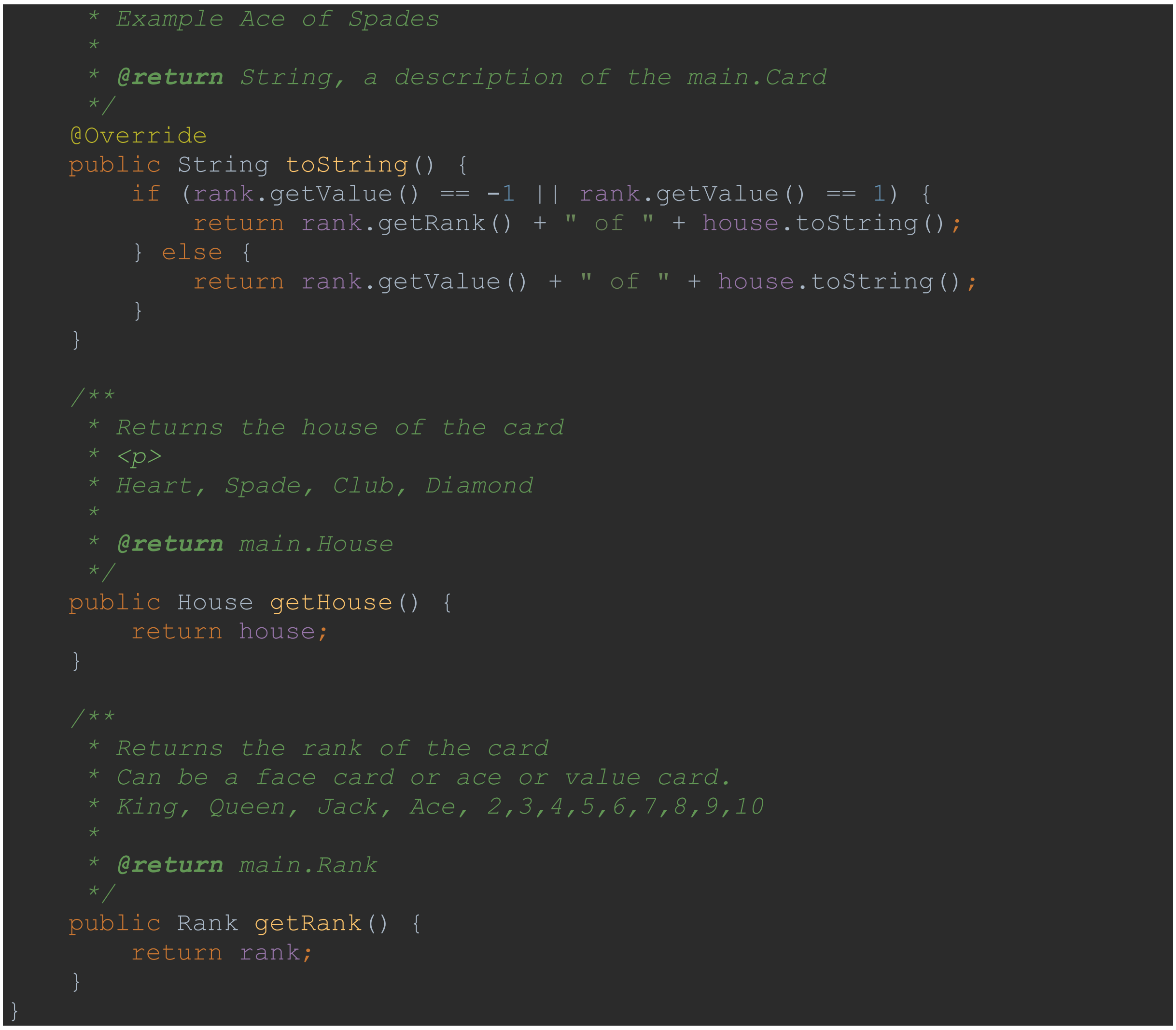
Java Classes

Card.java

```
package main;

/**
 * This Class is used to represent a Card with this application.
 * Contains information about its house and its rank.
 */
public class Card extends Colors {

    private House house;
    private Rank rank;

    /**
     * Constructor for an object representing a playing main.Card.
     * Only require a constructor with both fields, as a card
     * must have a main.House and a main.Rank in main.Elevens.
     * <p>
     * No field should ever be defaulted in this class.
     *
     * @param house house/suite of the card, Enum of main.House
     * @param rank  rank of the card, Enum of main.Rank: King, Queen, Jack,
Ace, One,Two,Three,Four,Five,Six,Seven,Nine,Ten
     */

    public Card(House house, Rank rank) {
        this.house = house;
        this.rank = rank;
    }

    /**
     * Converts the Cards Rank Value into a single or double letter digit.
     *
     * @param aCard
     * @return String, single letter String value of main.House
     */
    public static String extractRankAsDigit(Card aCard) {
        String output = "";
        switch (aCard.getRank()) {
            case KING:
                output = "K";
                break;
            case QUEEN:
                output = "Q";
                break;
            case JACK:
                output = "J";
                break;
            case ACE:
                output = "A";
                break;
            case TWO:
                output = "2";
                break;
            case THREE:
                output = "3";
```



```
                break;
            case FOUR:
                output = "4";
                break;
            case FIVE:
                output = "5";
                break;
            case SIX:
                output = "6";
                break;
            case SEVEN:
                output = "7";
                break;
            case EIGHT:
                output = "8";
                break;
            case NINE:
                output = "9";
                break;
            case TEN:
                output = "10";
                break;
        }
        return output;
    }

    /**
     * Converts A Cards House Object into a Single Color coded Digit.
     *
     * @param aCard
     * @return String, single letter String value of main.House
     */
    public static String extractHouseAsDigitWithColor(Card aCard) {
        String output = "";
        switch (aCard.getHouse()) {
            case DIAMONDS:
                output = COLOR_RED + "D" + COLOR_WHITE;
                break;
            case HEARTS:
                output = COLOR_RED + "H" + COLOR_WHITE;
                break;
            case CLUBS:
                output = COLOR_GREEN + "C" + COLOR_WHITE;
                break;
            case SPADES:
                output = COLOR_GREEN + "S" + COLOR_WHITE;
                break;
        }
        return output;
    }

    /**
     * Information such as the cards main.Rank and main.House as a String.
     * <p>
     * If is a face card will return Rank as String and its house including
Ace(but is not a face card).
     * If is a value card will return numeric the value of the card and its
house.
     * <p>
     * Example King of DIAMONDS
     * Example 10 of DIAMONDS
```



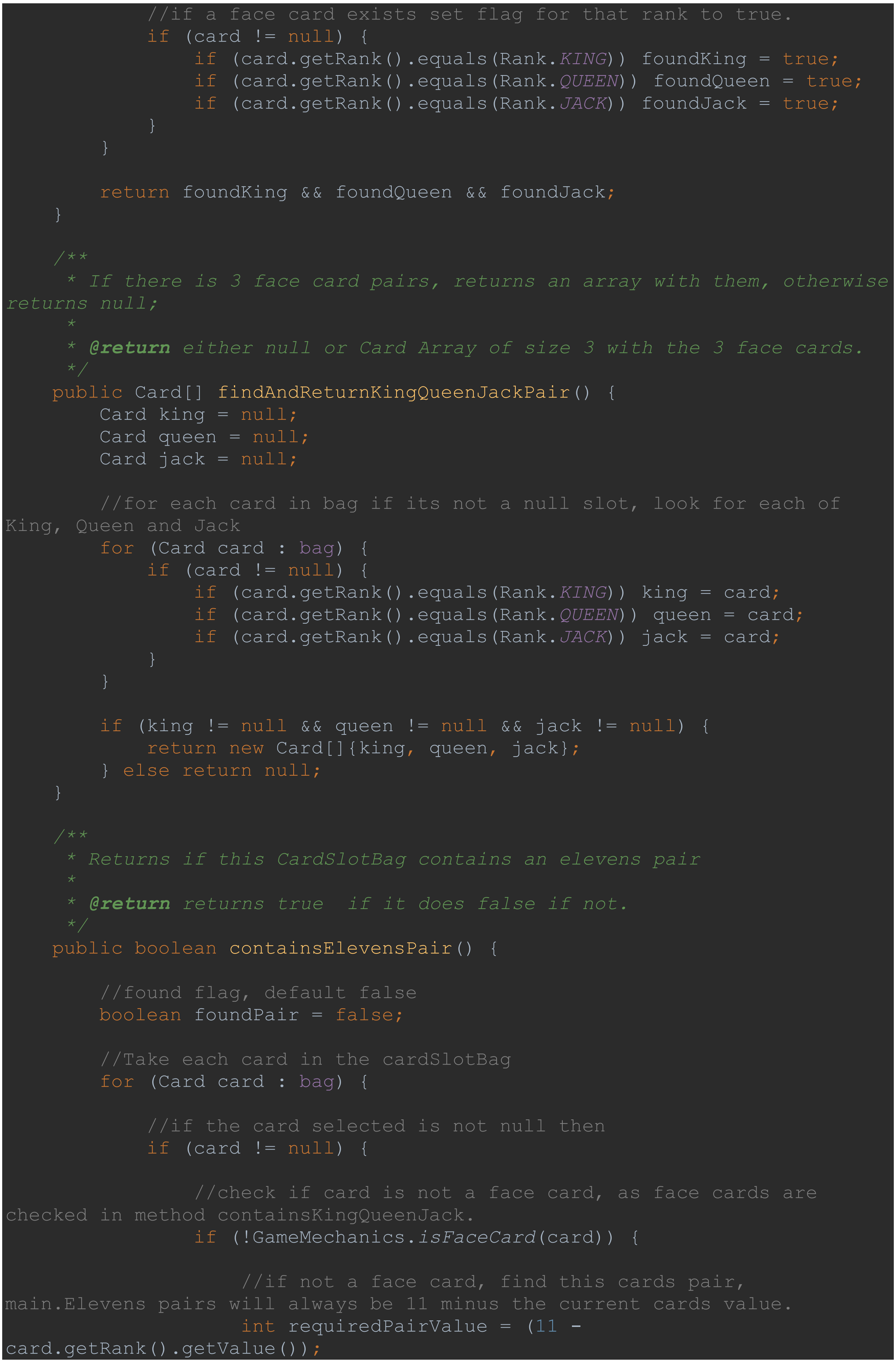
```
     * Example Ace of Spades
     *
     * @return String, a description of the main.Card
     */
    @Override
    public String toString() {
        if (rank.getValue() == -1 || rank.getValue() == 1) {
            return rank.getRank() + " of " + house.toString();
        } else {
            return rank.getValue() + " of " + house.toString();
        }
    }

    /**
     * Returns the house of the card
     * <p>
     * Heart, Spade, Club, Diamond
     *
     * @return main.House
     */
    public House getHouse() {
        return house;
    }

    /**
     * Returns the rank of the card
     * Can be a face card or ace or value card.
     * King, Queen, Jack, Ace, 2,3,4,5,6,7,8,9,10
     *
     * @return main.Rank
     */
    public Rank getRank() {
        return rank;
    }
}
```

CardNode.java

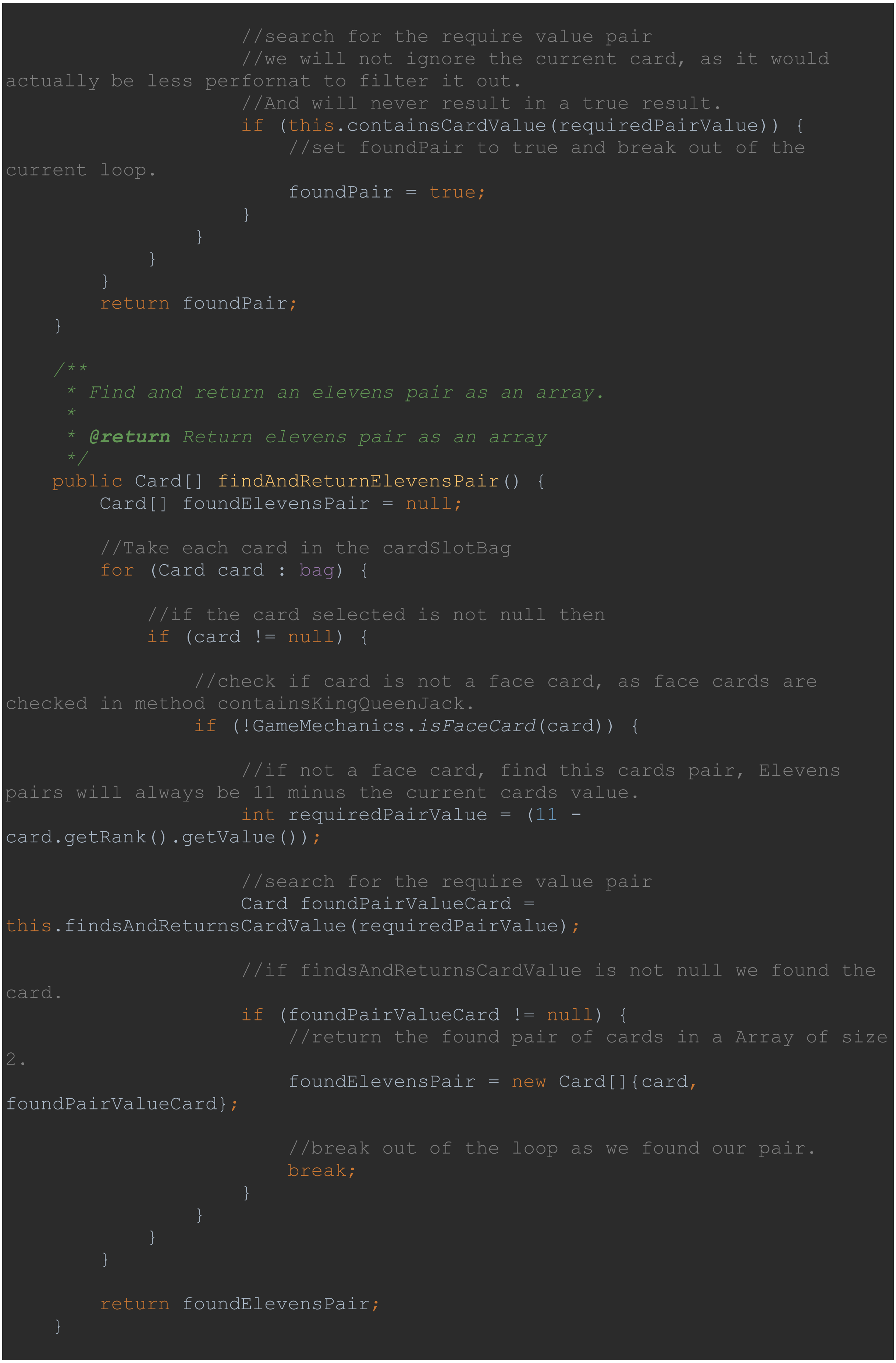
```java
package main;

/**
 * A class to as a wrapper object for a Card when being used in the Deck
Class
 * Allows Card to be Abstracted from an ADT.
 */
public class CardNode {

    private Card data;
    private CardNode next;

    /**
     * Constructor for CardNode
     *
     * @param dataValue Card held by the CardNode
     */
    public CardNode(Card dataValue) {
        this.data = dataValue;
        this.next = null;
    }

    /**
     * Return the Card held in the CardNode
     *
     * @return Card the card held in the node
     */
    public Card getData() {
        return data;
    }

    /**
     * Set the Card in the CardNode
     *
     * @param dataValue the Card you want to set as the CardNodes data
value.
     */
    public void setData(Card dataValue) {
        this.data = dataValue;
    }

    /**
     * Get the next CardNode linked to this CardNode
     *
     * @return CardNode
     */
    public CardNode getNext() {
        return next;
    }

    /**
     * Set the next CardNode
     *
     * @param nextNode set the next CardNode
     */
    public void setNext(CardNode nextNode) {
        this.next = nextNode;
    }
}
```

CardSlotsBag.java

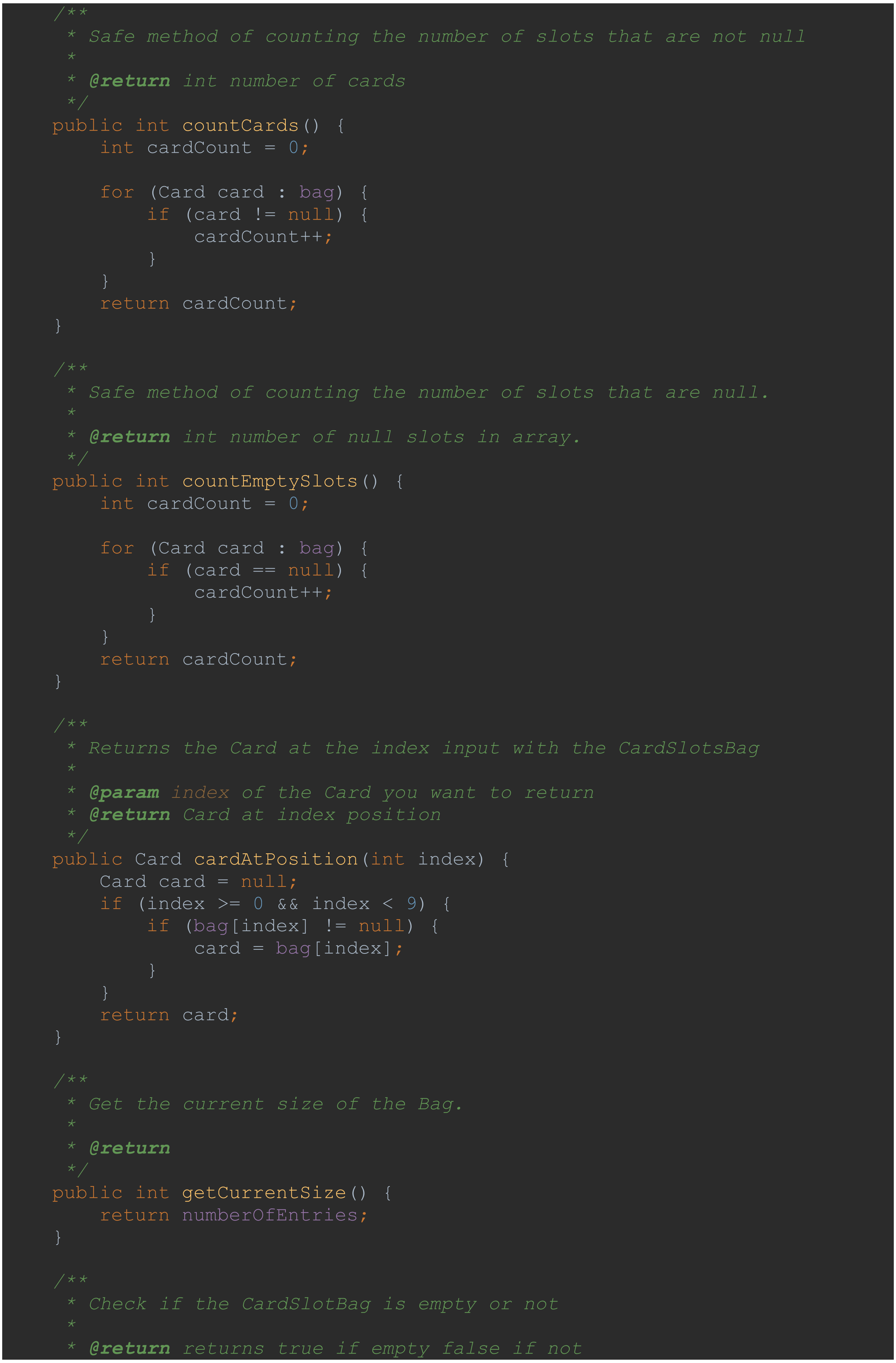
```java
package main;

import Interfaces.BagInterface;

public final class CardSlotsBag implements BagInterface<Card> {

    private static final int MAX_CAPACITY = 9;
    private Card[] bag;
    private int numberOfEntries;

    /**
     * Constructor for CardSlotsBag
     * Creates and Empty Bag
     */
    public CardSlotsBag() {
        bag = new Card[MAX_CAPACITY];
        numberOfEntries = 0;
    }

    /**
     * Creates a cardSlot Bag from the input CardSlotsBag
     *
     * @param copiedBag
     */
    public CardSlotsBag(Card[] copiedBag) {
        bag = copiedBag;
        numberOfEntries = countCards();
    }

    /**
     * Used to format card values if single digit or double digit.
     * <p>
     * used to format Strings for display method.
     *
     * @param str string to format
     * @return returns the formatted string.
     */
    private static String formatStringForDisplay(String str) {
        try {
            int parsedInt = Integer.parseInt(str);

            //dont append extra whitespace if double digit int value.
            if (parsedInt > 9) return parsedInt + "";
            else return parsedInt + " ";
        } catch (Exception e) {
            //cant convert to string so much be a single letter either,
Jack King Queen or Ace.
            return str + " ";
        }
    }

    /**
     * Check if the current CardSlotBag contains a card with the value of
the input int.
     *
     * @param cardValue the card value
     * @return true if card with value is found and false if not.
     */
    public boolean containsCardValue(int cardValue) {
        boolean found = false;
        int index = 0;

        while (!found && index < bag.length) {

            if (bag[index] != null) {

                if (bag[index].getRank().getValue() == cardValue) {

                    if (!GameMechanics.isFaceCard(bag[index])) {
                        found = true;
                    }
                }
            }
            index++;
        }
        return found;
    }

    /**
     * Returns the card with the input card from the current card
CardsSlotBag.
     *
     * @param cardValue find the card with the input value and return it.
     * @return Card to
     */
    public Card findsAndReturnsCardValue(int cardValue) {
        Card foundCard = null;
        int index = 0;

        while (index < bag.length) {

            if (bag[index] != null) {

                if (bag[index].getRank().getValue() == cardValue) {

                    if (!GameMechanics.isFaceCard(bag[index])) {
                        foundCard = bag[index];
                    }
                }
            }
            index++;
        }

        return foundCard;
    }

    /**
     * Returns true if this contains 3 face cards of jack, queen and king.
     *
     * @return boolean if all 3 are found.
     */
    public boolean containsKingQueenJack() {
        boolean foundKing = false;
        boolean foundQueen = false;
        boolean foundJack = false;

        for (Card card : bag) {

            //if the current card is not null, check for each of the face
cards.
```



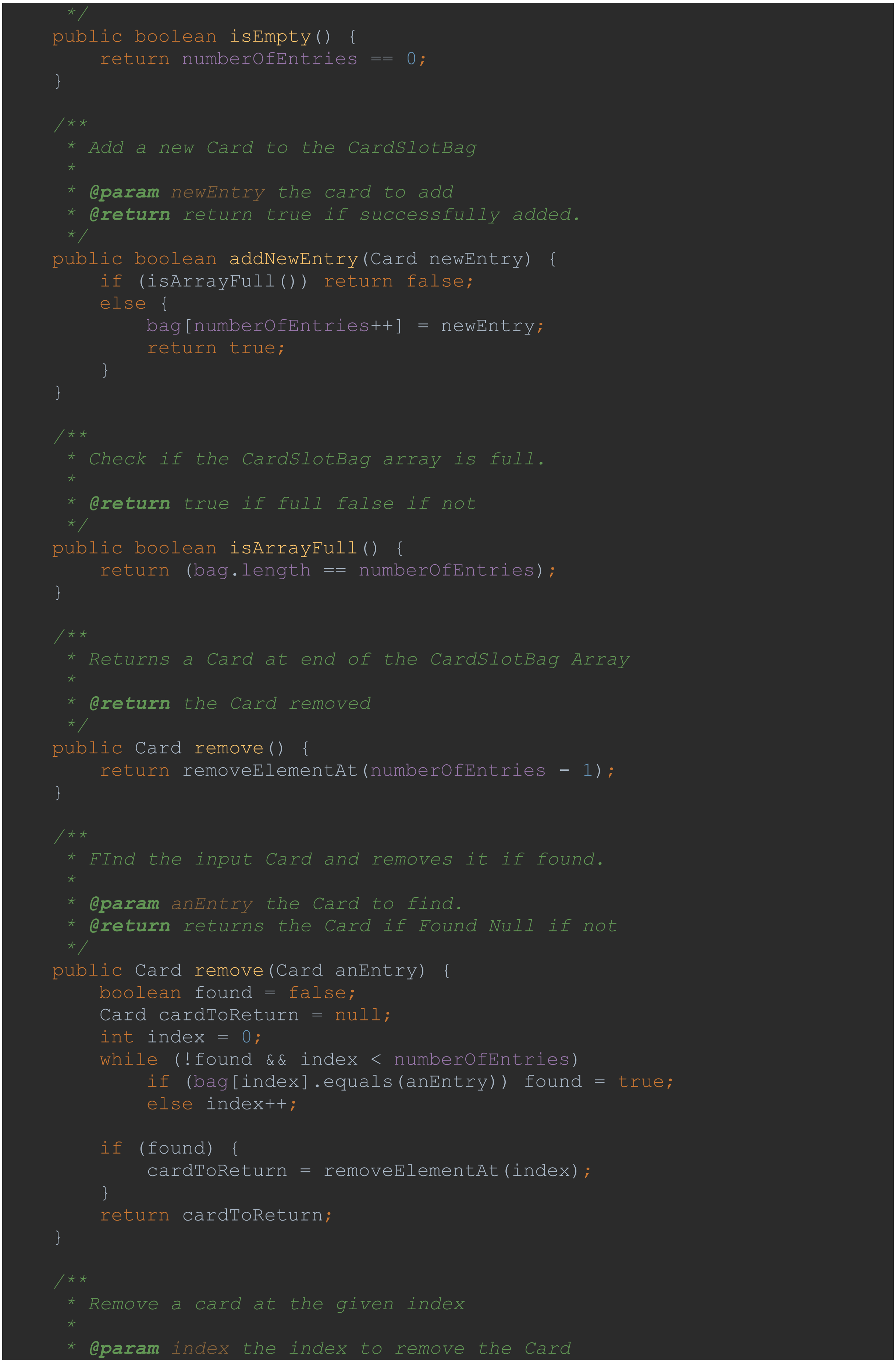
```
            //if a face card exists set flag for that rank to true.
            if (card != null) {
                if (card.getRank().equals(Rank.KING)) foundKing = true;
                if (card.getRank().equals(Rank.QUEEN)) foundQueen = true;
                if (card.getRank().equals(Rank.JACK)) foundJack = true;
            }
        }

        return foundKing && foundQueen && foundJack;
    }

    /**
     * If there is 3 face card pairs, returns an array with them, otherwise
returns null;
     *
     * @return either null or Card Array of size 3 with the 3 face cards.
     */
    public Card[] findAndReturnKingQueenJackPair() {
        Card king = null;
        Card queen = null;
        Card jack = null;

        //for each card in bag if its not a null slot, look for each of
King, Queen and Jack
        for (Card card : bag) {
            if (card != null) {
                if (card.getRank().equals(Rank.KING)) king = card;
                if (card.getRank().equals(Rank.QUEEN)) queen = card;
                if (card.getRank().equals(Rank.JACK)) jack = card;
            }
        }

        if (king != null && queen != null && jack != null) {
            return new Card[]{king, queen, jack};
        } else return null;
    }

    /**
     * Returns if this CardSlotBag contains an elevens pair
     *
     * @return returns true  if it does false if not.
     */
    public boolean containsElevensPair() {

        //found flag, default false
        boolean foundPair = false;

        //Take each card in the cardSlotBag
        for (Card card : bag) {

            //if the card selected is not null then
            if (card != null) {

                //check if card is not a face card, as face cards are
checked in method containsKingQueenJack.
                if (!GameMechanics.isFaceCard(card)) {

                    //if not a face card, find this cards pair,
main.Elevens pairs will always be 11 minus the current cards value.
                    int requiredPairValue = (11 -
card.getRank().getValue());
```



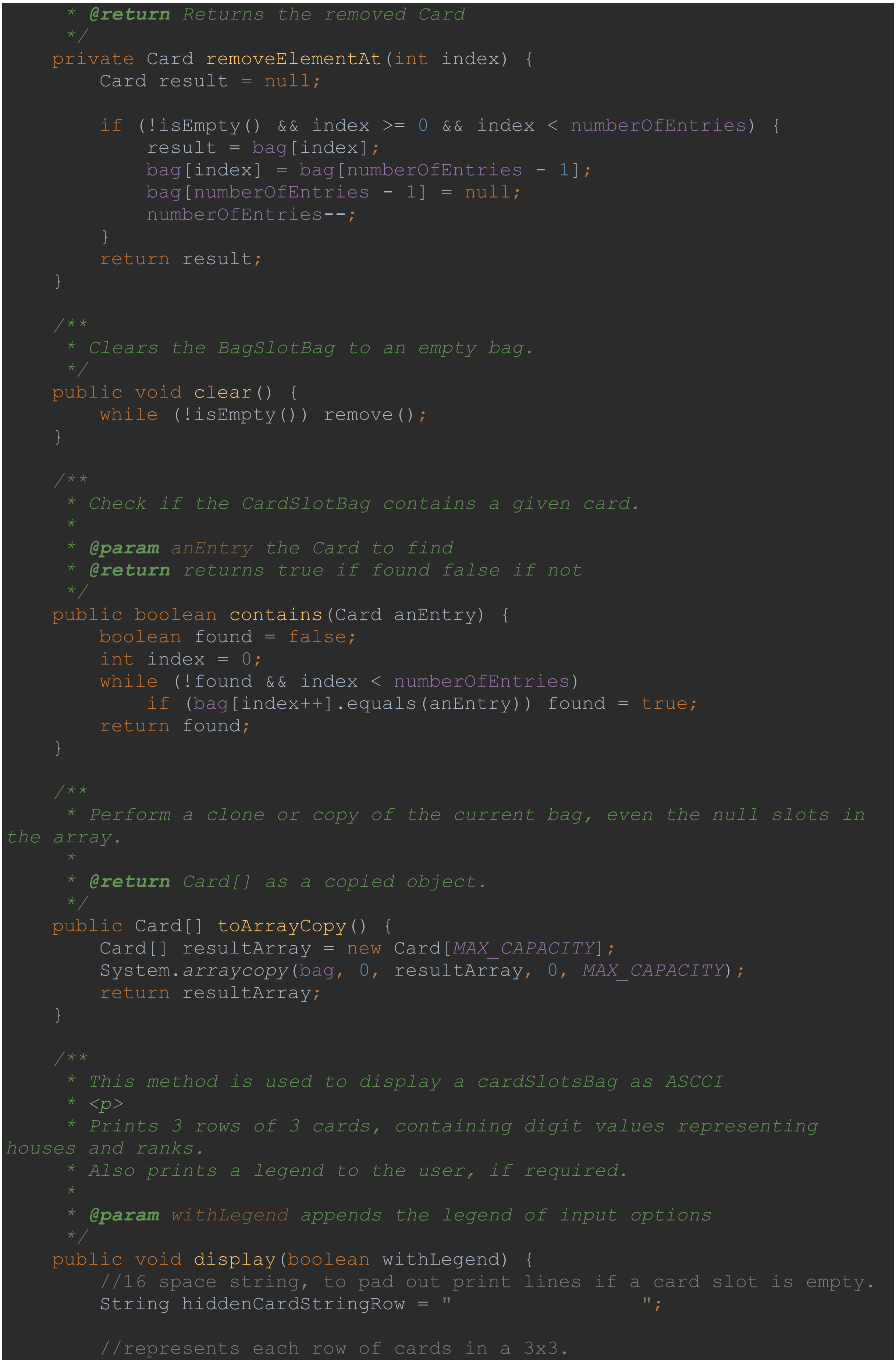
```
                    //search for the require value pair
                    //we will not ignore the current card, as it would
actually be less perfornat to filter it out.
                    //And will never result in a true result.
                    if (this.containsCardValue(requiredPairValue)) {
                        //set foundPair to true and break out of the
current loop.
                        foundPair = true;
                    }
                }
            }
        }
        return foundPair;
    }

    /**
     * Find and return an elevens pair as an array.
     *
     * @return Return elevens pair as an array
     */
    public Card[] findAndReturnElevensPair() {
        Card[] foundElevensPair = null;

        //Take each card in the cardSlotBag
        for (Card card : bag) {

            //if the card selected is not null then
            if (card != null) {

                //check if card is not a face card, as face cards are
checked in method containsKingQueenJack.
                if (!GameMechanics.isFaceCard(card)) {

                    //if not a face card, find this cards pair, Elevens
pairs will always be 11 minus the current cards value.
                    int requiredPairValue = (11 -
card.getRank().getValue());

                    //search for the require value pair
                    Card foundPairValueCard =
this.findsAndReturnsCardValue(requiredPairValue);

                    //if findsAndReturnsCardValue is not null we found the
card.
                    if (foundPairValueCard != null) {
                        //return the found pair of cards in a Array of size
2.
                        foundElevensPair = new Card[]{card,
foundPairValueCard};

                        //break out of the loop as we found our pair.
                        break;
                    }
                }
            }
        }

        return foundElevensPair;
    }
```



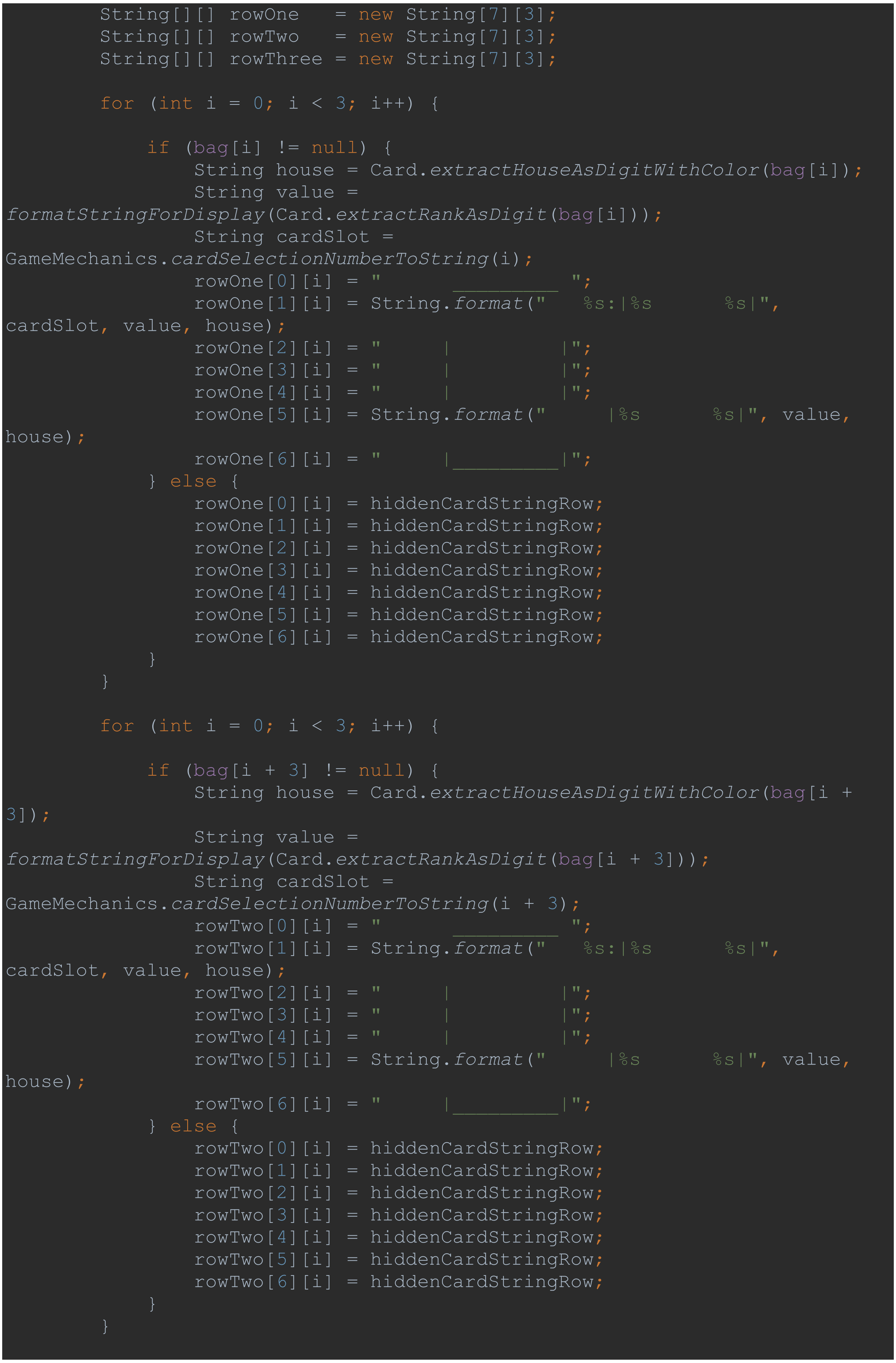
```java
    /**
     * Safe method of counting the number of slots that are not null
     *
     * @return int number of cards
     */
    public int countCards() {
        int cardCount = 0;

        for (Card card : bag) {
            if (card != null) {
                cardCount++;
            }
        }
        return cardCount;
    }

    /**
     * Safe method of counting the number of slots that are null.
     *
     * @return int number of null slots in array.
     */
    public int countEmptySlots() {
        int cardCount = 0;

        for (Card card : bag) {
            if (card == null) {
                cardCount++;
            }
        }
        return cardCount;
    }

    /**
     * Returns the Card at the index input with the CardSlotsBag
     *
     * @param index of the Card you want to return
     * @return Card at index position
     */
    public Card cardAtPosition(int index) {
        Card card = null;
        if (index >= 0 && index < 9) {
            if (bag[index] != null) {
                card = bag[index];
            }
        }
        return card;
    }

    /**
     * Get the current size of the Bag.
     *
     * @return
     */
    public int getCurrentSize() {
        return numberOfEntries;
    }

    /**
     * Check if the CardSlotBag is empty or not
     *
     * @return returns true if empty false if not
```



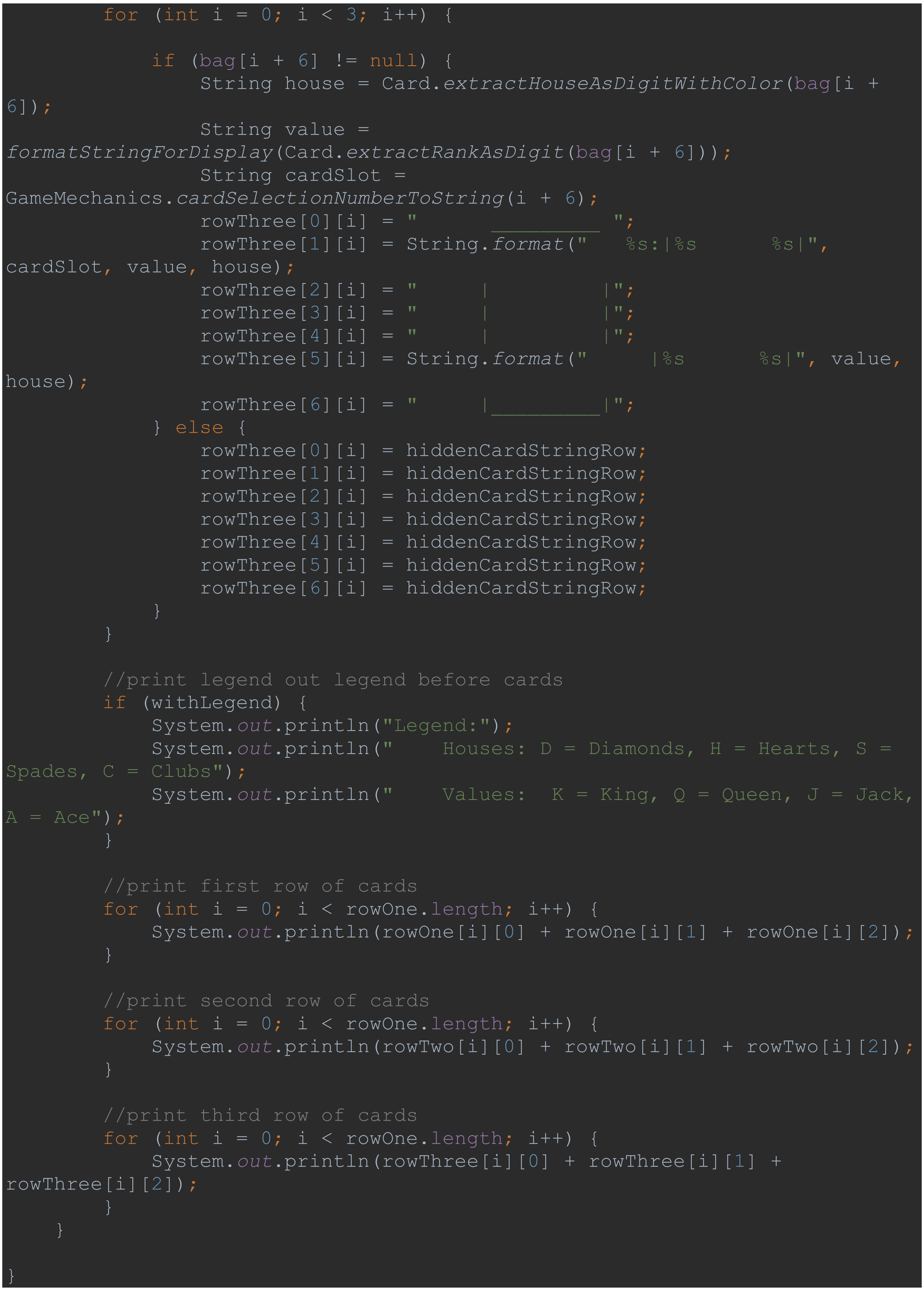
```
     */
    public boolean isEmpty() {
        return numberOfEntries == 0;
    }

    /**
     * Add a new Card to the CardSlotBag
     *
     * @param newEntry the card to add
     * @return return true if successfully added.
     */
    public boolean addNewEntry(Card newEntry) {
        if (isArrayFull()) return false;
        else {
            bag[numberOfEntries++] = newEntry;
            return true;
        }
    }

    /**
     * Check if the CardSlotBag array is full.
     *
     * @return true if full false if not
     */
    public boolean isArrayFull() {
        return (bag.length == numberOfEntries);
    }

    /**
     * Returns a Card at end of the CardSlotBag Array
     *
     * @return the Card removed
     */
    public Card remove() {
        return removeElementAt(numberOfEntries - 1);
    }

    /**
     * FInd the input Card and removes it if found.
     *
     * @param anEntry the Card to find.
     * @return returns the Card if Found Null if not
     */
    public Card remove(Card anEntry) {
        boolean found = false;
        Card cardToReturn = null;
        int index = 0;
        while (!found && index < numberOfEntries)
            if (bag[index].equals(anEntry)) found = true;
            else index++;

        if (found) {
            cardToReturn = removeElementAt(index);
        }
        return cardToReturn;
    }

    /**
     * Remove a card at the given index
     *
     * @param index the index to remove the Card
```



```
     * @return Returns the removed Card
     */
    private Card removeElementAt(int index) {
        Card result = null;

        if (!isEmpty() && index >= 0 && index < numberOfEntries) {
            result = bag[index];
            bag[index] = bag[numberOfEntries - 1];
            bag[numberOfEntries - 1] = null;
            numberOfEntries--;
        }
        return result;
    }

    /**
     * Clears the BagSlotBag to an empty bag.
     */
    public void clear() {
        while (!isEmpty()) remove();
    }

    /**
     * Check if the CardSlotBag contains a given card.
     *
     * @param anEntry the Card to find
     * @return returns true if found false if not
     */
    public boolean contains(Card anEntry) {
        boolean found = false;
        int index = 0;
        while (!found && index < numberOfEntries)
            if (bag[index++].equals(anEntry)) found = true;
        return found;
    }

    /**
     * Perform a clone or copy of the current bag, even the null slots in
the array.
     *
     * @return Card[] as a copied object.
     */
    public Card[] toArrayCopy() {
        Card[] resultArray = new Card[MAX_CAPACITY];
        System.arraycopy(bag, 0, resultArray, 0, MAX_CAPACITY);
        return resultArray;
    }

    /**
     * This method is used to display a cardSlotsBag as ASCCI
     * <p>
     * Prints 3 rows of 3 cards, containing digit values representing
houses and ranks.
     * Also prints a legend to the user, if required.
     *
     * @param withLegend appends the legend of input options
     */
    public void display(boolean withLegend) {
        //16 space string, to pad out print lines if a card slot is empty.
        String hiddenCardStringRow = "                ";

        //represents each row of cards in a 3x3.
```



```
        String[][] rowOne   = new String[7][3];
        String[][] rowTwo   = new String[7][3];
        String[][] rowThree = new String[7][3];

        for (int i = 0; i < 3; i++) {

            if (bag[i] != null) {
                String house = Card.extractHouseAsDigitWithColor(bag[i]);
                String value =
formatStringForDisplay(Card.extractRankAsDigit(bag[i]));
                String cardSlot =
GameMechanics.cardSelectionNumberToString(i);
                rowOne[0][i] = "      _________ ";
                rowOne[1][i] = String.format("   %s:|%s      %s|",
cardSlot, value, house);
                rowOne[2][i] = "     |         |";
                rowOne[3][i] = "     |         |";
                rowOne[4][i] = "     |         |";
                rowOne[5][i] = String.format("     |%s      %s|", value,
house);
                rowOne[6][i] = "     |_________|";
            } else {
                rowOne[0][i] = hiddenCardStringRow;
                rowOne[1][i] = hiddenCardStringRow;
                rowOne[2][i] = hiddenCardStringRow;
                rowOne[3][i] = hiddenCardStringRow;
                rowOne[4][i] = hiddenCardStringRow;
                rowOne[5][i] = hiddenCardStringRow;
                rowOne[6][i] = hiddenCardStringRow;
            }
        }

        for (int i = 0; i < 3; i++) {

            if (bag[i + 3] != null) {
                String house = Card.extractHouseAsDigitWithColor(bag[i +
3]);
                String value =
formatStringForDisplay(Card.extractRankAsDigit(bag[i + 3]));
                String cardSlot =
GameMechanics.cardSelectionNumberToString(i + 3);
                rowTwo[0][i] = "      _________ ";
                rowTwo[1][i] = String.format("   %s:|%s      %s|",
cardSlot, value, house);
                rowTwo[2][i] = "     |         |";
                rowTwo[3][i] = "     |         |";
                rowTwo[4][i] = "     |         |";
                rowTwo[5][i] = String.format("     |%s      %s|", value,
house);
                rowTwo[6][i] = "     |_________|";
            } else {
                rowTwo[0][i] = hiddenCardStringRow;
                rowTwo[1][i] = hiddenCardStringRow;
                rowTwo[2][i] = hiddenCardStringRow;
                rowTwo[3][i] = hiddenCardStringRow;
                rowTwo[4][i] = hiddenCardStringRow;
                rowTwo[5][i] = hiddenCardStringRow;
                rowTwo[6][i] = hiddenCardStringRow;
            }
        }
```



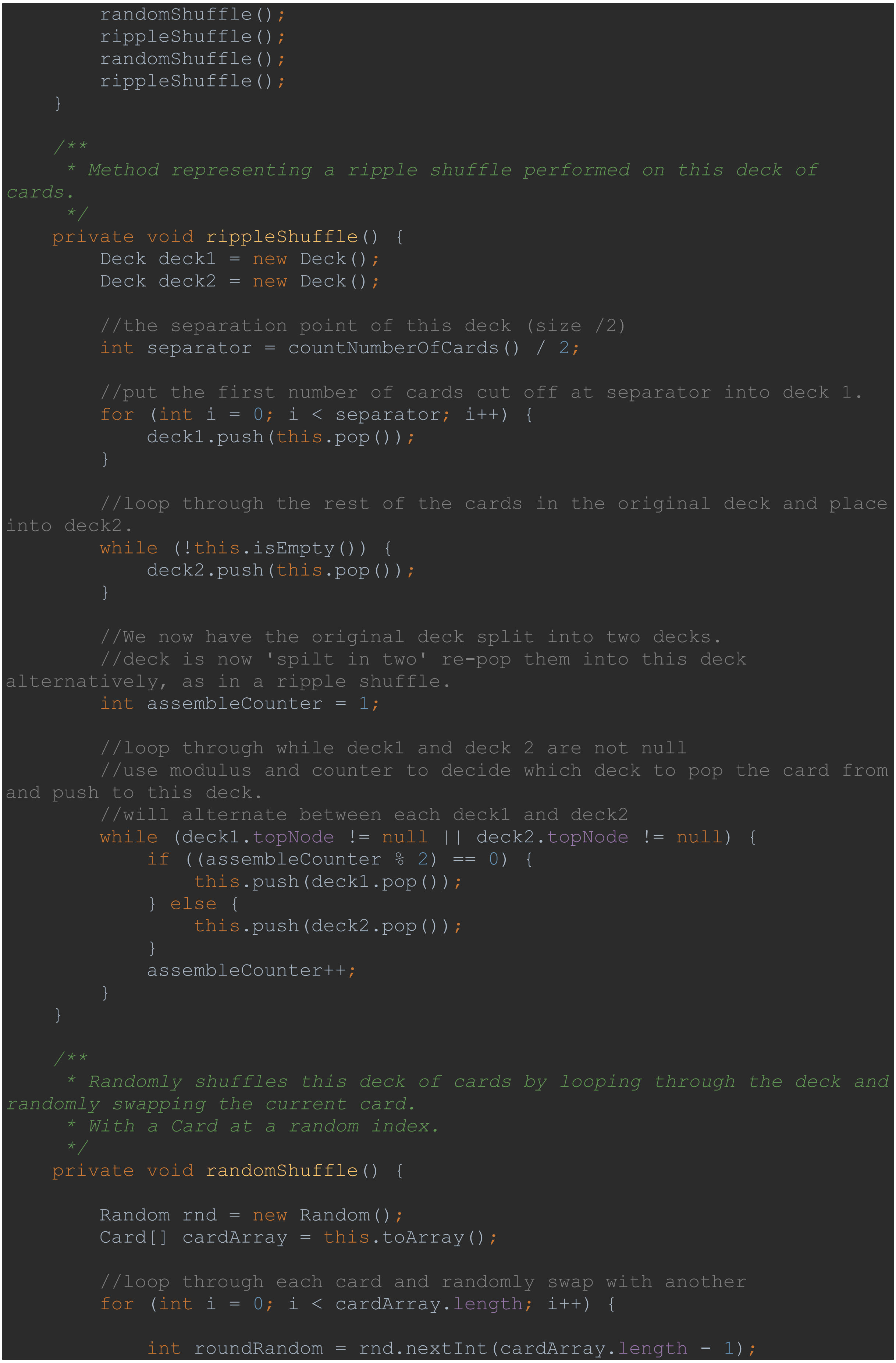
```
        for (int i = 0; i < 3; i++) {

            if (bag[i + 6] != null) {
                String house = Card.extractHouseAsDigitWithColor(bag[i +
6]);
                String value =
formatStringForDisplay(Card.extractRankAsDigit(bag[i + 6]));
                String cardSlot =
GameMechanics.cardSelectionNumberToString(i + 6);
                rowThree[0][i] = "      _________ ";
                rowThree[1][i] = String.format("   %s:|%s      %s|",
cardSlot, value, house);
                rowThree[2][i] = "     |         |";
                rowThree[3][i] = "     |         |";
                rowThree[4][i] = "     |         |";
                rowThree[5][i] = String.format("     |%s      %s|", value,
house);
                rowThree[6][i] = "     |_________|";
            } else {
                rowThree[0][i] = hiddenCardStringRow;
                rowThree[1][i] = hiddenCardStringRow;
                rowThree[2][i] = hiddenCardStringRow;
                rowThree[3][i] = hiddenCardStringRow;
                rowThree[4][i] = hiddenCardStringRow;
                rowThree[5][i] = hiddenCardStringRow;
                rowThree[6][i] = hiddenCardStringRow;
            }
        }

        //print legend out legend before cards
        if (withLegend) {
            System.out.println("Legend:");
            System.out.println("    Houses: D = Diamonds, H = Hearts, S =
Spades, C = Clubs");
            System.out.println("    Values:  K = King, Q = Queen, J = Jack,
A = Ace");
        }

        //print first row of cards
        for (int i = 0; i < rowOne.length; i++) {
            System.out.println(rowOne[i][0] + rowOne[i][1] + rowOne[i][2]);
        }

        //print second row of cards
        for (int i = 0; i < rowOne.length; i++) {
            System.out.println(rowTwo[i][0] + rowTwo[i][1] + rowTwo[i][2]);
        }

        //print third row of cards
        for (int i = 0; i < rowOne.length; i++) {
            System.out.println(rowThree[i][0] + rowThree[i][1] +
rowThree[i][2]);
        }
    }

}
```

Colors.java

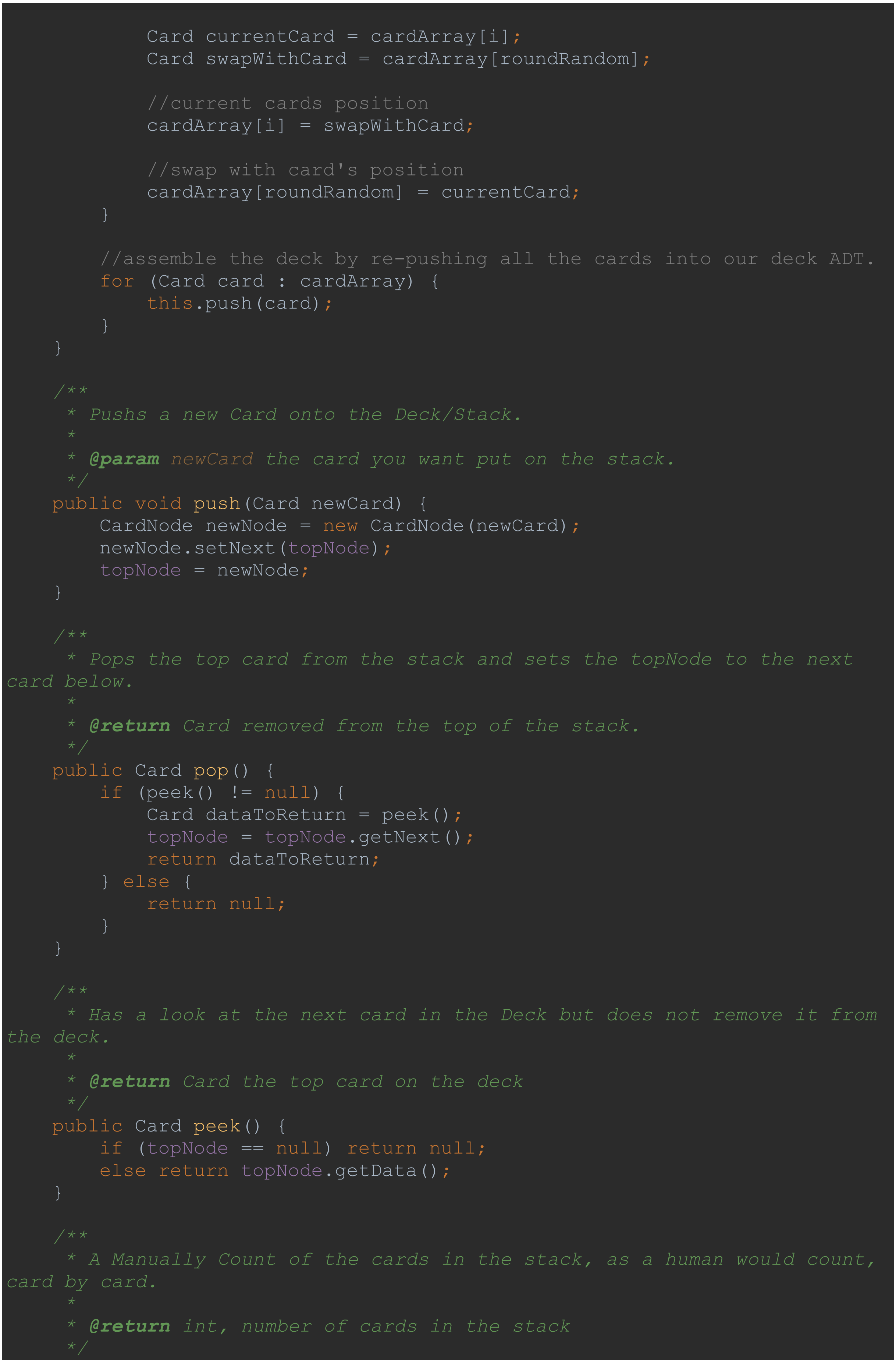
```java
package main;

/**
 * A class that can be inherited to provide colors to string before
system.out.
 */
public class Colors {
    static final String COLOR_RED = "\u001B[31m";
    static final String COLOR_GREEN = "\u001B[32m";
    static final String COLOR_WHITE = "\u001B[0m";
}
```

Deck.java

```java
package main;

import Interfaces.StackInterface;

import java.util.Random;

public class Deck implements StackInterface<Card> {

    private CardNode topNode;

    /**
     * Used to populate this deck with all the required cards in a 52 card
deck.
     */
    public void createFullDeckOfCards() {
        //Pack of card typically comes in the reverse of this order, but
will be correct when popping from the stack.
        House[] houseArray = {House.SPADES, House.DIAMONDS, House.CLUBS,
House.HEARTS};

        //in reverse order, as when we push the order will be flipped if we
popped all cards.
        Rank[] rankArray = {
                Rank.KING, Rank.QUEEN, Rank.JACK, Rank.TEN, Rank.NINE,
                Rank.EIGHT, Rank.SEVEN, Rank.SIX, Rank.FIVE, Rank.FOUR,
                Rank.THREE, Rank.TWO, Rank.ACE
        };

        //for each house, loop through each rank and push that card to the
stack.
        for (House house : houseArray) {
            for (Rank rank : rankArray) {
                this.push(new Card(house, rank));
            }
        }
    }

    /**
     * A number of shuffle to make sure the cards are well shuffled
followed by another ripple
     * To make sure the cards are well shuffled.
     */
    public void rigorousShuffle() {
        rippleShuffle();
```



```
        randomShuffle();
        rippleShuffle();
        randomShuffle();
        rippleShuffle();
    }

    /**
     * Method representing a ripple shuffle performed on this deck of
cards.
     */
    private void rippleShuffle() {
        Deck deck1 = new Deck();
        Deck deck2 = new Deck();

        //the separation point of this deck (size /2)
        int separator = countNumberOfCards() / 2;

        //put the first number of cards cut off at separator into deck 1.
        for (int i = 0; i < separator; i++) {
            deck1.push(this.pop());
        }

        //loop through the rest of the cards in the original deck and place
into deck2.
        while (!this.isEmpty()) {
            deck2.push(this.pop());
        }

        //We now have the original deck split into two decks.
        //deck is now 'spilt in two' re-pop them into this deck
alternatively, as in a ripple shuffle.
        int assembleCounter = 1;

        //loop through while deck1 and deck 2 are not null
        //use modulus and counter to decide which deck to pop the card from
and push to this deck.
        //will alternate between each deck1 and deck2
        while (deck1.topNode != null || deck2.topNode != null) {
            if ((assembleCounter % 2) == 0) {
                this.push(deck1.pop());
            } else {
                this.push(deck2.pop());
            }
            assembleCounter++;
        }
    }

    /**
     * Randomly shuffles this deck of cards by looping through the deck and
randomly swapping the current card.
     * With a Card at a random index.
     */
    private void randomShuffle() {

        Random rnd = new Random();
        Card[] cardArray = this.toArray();

        //loop through each card and randomly swap with another
        for (int i = 0; i < cardArray.length; i++) {

            int roundRandom = rnd.nextInt(cardArray.length - 1);
```

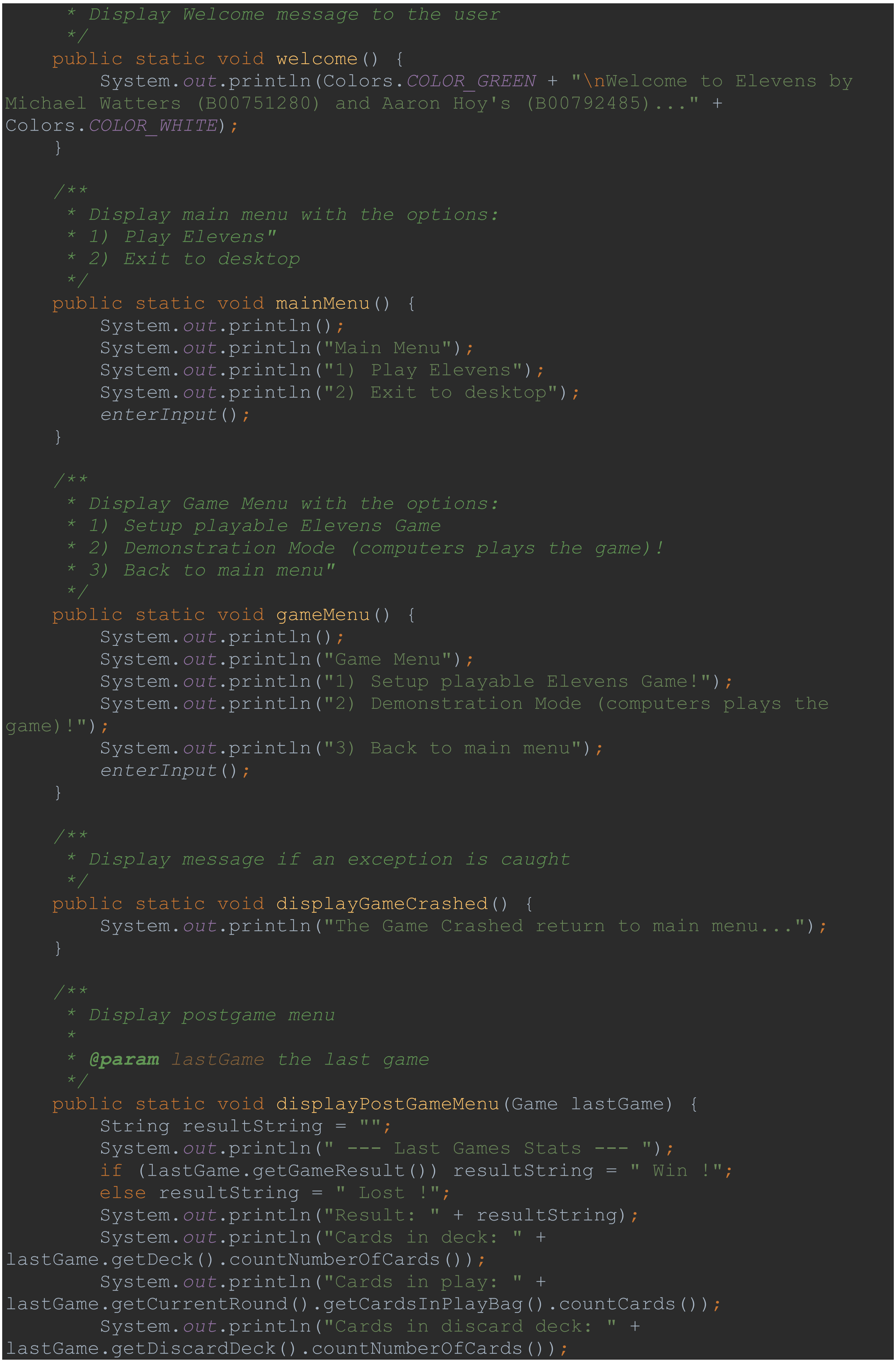


```
            Card currentCard = cardArray[i];
            Card swapWithCard = cardArray[roundRandom];

            //current cards position
            cardArray[i] = swapWithCard;

            //swap with card's position
            cardArray[roundRandom] = currentCard;
        }

        //assemble the deck by re-pushing all the cards into our deck ADT.
        for (Card card : cardArray) {
            this.push(card);
        }
    }

    /**
     * Pushs a new Card onto the Deck/Stack.
     *
     * @param newCard the card you want put on the stack.
     */
    public void push(Card newCard) {
        CardNode newNode = new CardNode(newCard);
        newNode.setNext(topNode);
        topNode = newNode;
    }

    /**
     * Pops the top card from the stack and sets the topNode to the next
card below.
     *
     * @return Card removed from the top of the stack.
     */
    public Card pop() {
        if (peek() != null) {
            Card dataToReturn = peek();
            topNode = topNode.getNext();
            return dataToReturn;
        } else {
            return null;
        }
    }

    /**
     * Has a look at the next card in the Deck but does not remove it from
the deck.
     *
     * @return Card the top card on the deck
     */
    public Card peek() {
        if (topNode == null) return null;
        else return topNode.getData();
    }

    /**
     * A Manually Count of the cards in the stack, as a human would count,
card by card.
     *
     * @return int, number of cards in the stack
     */
```

```java
    public int countNumberOfCards() {
        int count = 0;
        if (topNode == null) {
            return 0;
        } else {
            CardNode currentNode = topNode;
            while (currentNode != null) {
                currentNode = currentNode.getNext();
                count++;
            }
        }
        return count;
    }

    /**
     * Checks if the stack is empty
     *
     * @return boolean returns true if empty
     */
    public boolean isEmpty() {
        return (topNode == null);
    }

    /**
     * Clears the deck/stack.
     */
    public void clear() {
        topNode = null;
    }

    /**
     * Converts the Deck/Stack into an Array by popping, the stack will be empty after this method is used.
     *
     * @return Card[] converts the Stack to an Array
     */
    public Card[] toArray() {
        Card[] cardArray = new Card[countNumberOfCards()];

        for (int i = 0; i < cardArray.length; i++) {
            cardArray[i] = this.pop();
        }

        return cardArray;
    }
}
```

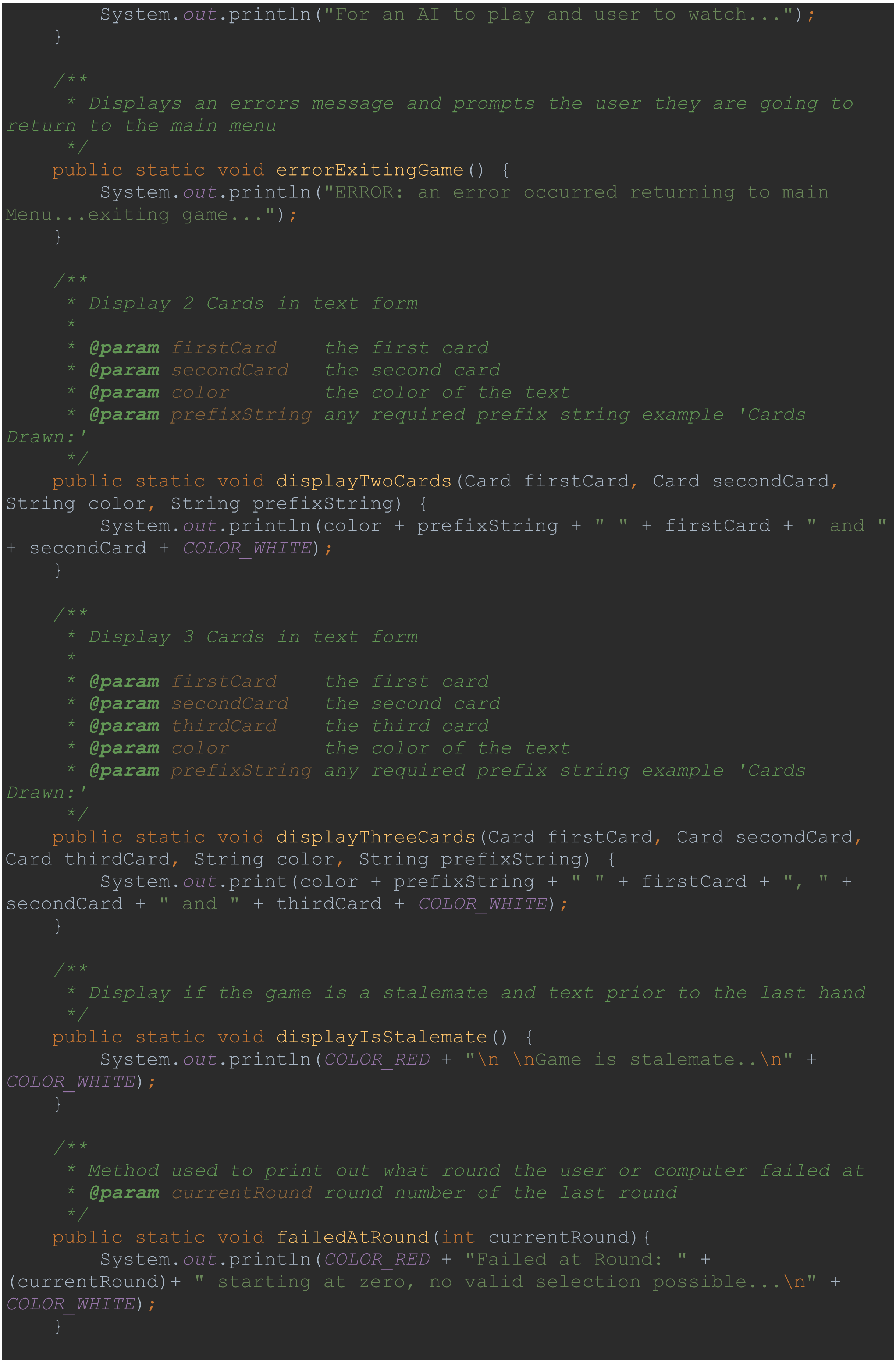
Display.java

```java
package main;

import java.util.Scanner;

/**
 * This class is used to abstract system.out from code blocks,
 * for readability.
 */
public class Display extends Colors {

    /**
```

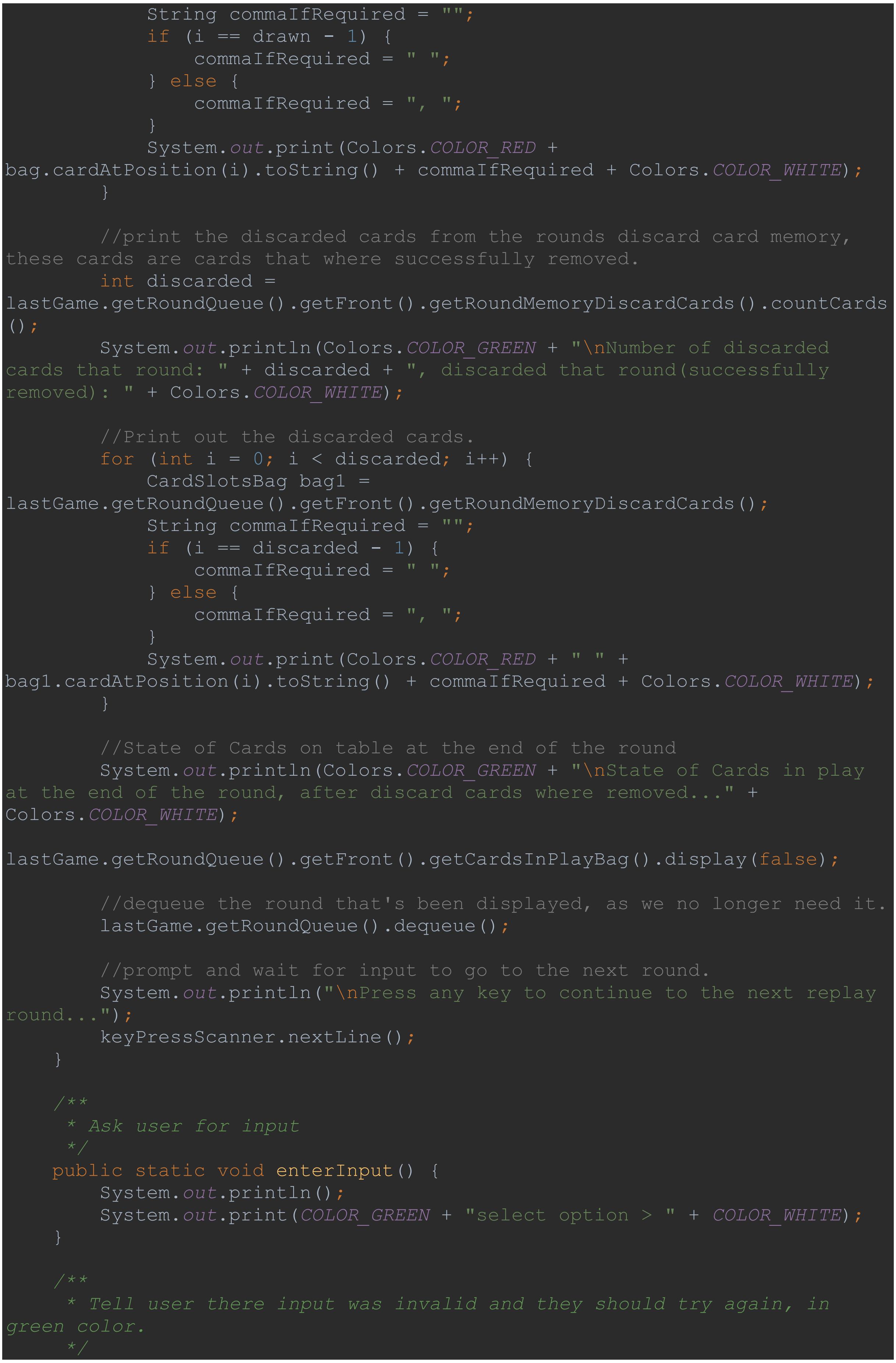


```
     * Display Welcome message to the user
     */
    public static void welcome() {
        System.out.println(Colors.COLOR_GREEN + "\nWelcome to Elevens by
Michael Watters (B00751280) and Aaron Hoy's (B00792485)..." +
Colors.COLOR_WHITE);
    }

    /**
     * Display main menu with the options:
     * 1) Play Elevens"
     * 2) Exit to desktop
     */
    public static void mainMenu() {
        System.out.println();
        System.out.println("Main Menu");
        System.out.println("1) Play Elevens");
        System.out.println("2) Exit to desktop");
        enterInput();
    }

    /**
     * Display Game Menu with the options:
     * 1) Setup playable Elevens Game
     * 2) Demonstration Mode (computers plays the game)!
     * 3) Back to main menu"
     */
    public static void gameMenu() {
        System.out.println();
        System.out.println("Game Menu");
        System.out.println("1) Setup playable Elevens Game!");
        System.out.println("2) Demonstration Mode (computers plays the
game)!");
        System.out.println("3) Back to main menu");
        enterInput();
    }

    /**
     * Display message if an exception is caught
     */
    public static void displayGameCrashed() {
        System.out.println("The Game Crashed return to main menu...");
    }

    /**
     * Display postgame menu
     *
     * @param lastGame the last game
     */
    public static void displayPostGameMenu(Game lastGame) {
        String resultString = "";
        System.out.println(" --- Last Games Stats --- ");
        if (lastGame.getGameResult()) resultString = " Win !";
        else resultString = " Lost !";
        System.out.println("Result: " + resultString);
        System.out.println("Cards in deck: " +
lastGame.getDeck().countNumberOfCards());
        System.out.println("Cards in play: " +
lastGame.getCurrentRound().getCardsInPlayBag().countCards());
        System.out.println("Cards in discard deck: " +
lastGame.getDiscardDeck().countNumberOfCards());
```

```
        System.out.println();
        System.out.println("Post Game Menu");
        System.out.println("1) Retry (play again)");
        System.out.println("2) Action Replay of the Last Games's Rounds!");
        System.out.println("3) Back to Game Menu...");
        enterInput();
    }

    /**
     * Display a round
     * 1) the round number
     * 2) the cards in play
     * 3) the input options legend
     *
     * @param currentRound the current round.
     */
    public static void displayRound(Round currentRound) {
        System.out.println();
        System.out.println("------------------------ Round " +
currentRound.getRoundNumber() + " ------------------------");
        currentRound.getCardsInPlayBag().display(true);
        System.out.println();
        System.out.println("Input Options:");
        System.out.println("    hint - displays a hint about cards to
pick.");
        System.out.println("    quit - quit to post game .");
        System.out.println("    valid card selection: a, b, c, d, e, f, g,
h, i");
        System.out.println("    select 2 cards: 'ab' for Elevens pair, or 3
cards: 'abc' for face Pairs.");
    }

    /**
     * Display Computer/demonstration modes round.
     *
     * @param currentRound current round
     */
    public static void displayAIRound(Round currentRound) {
        System.out.println();
        System.out.println("------------------------ Round " +
currentRound.getRoundNumber() + " ------------------------");
        currentRound.getCardsInPlayBag().display(true);
        System.out.println();
    }

    /**
     * Display setting up of game, for a human user.
     */
    public static void userPlayableGame() {
        System.out.println();
        System.out.println("Setting up game...");
        System.out.println("For a Human user...");
    }

    /**
     * Display setting up of game, for a non human user (demonstration
mode).
     */
    public static void aiPlayableGame() {
        System.out.println();
        System.out.println("Setting up game...");
```



```
        System.out.println("For an AI to play and user to watch...");
    }

    /**
     * Displays an errors message and prompts the user they are going to
return to the main menu
     */
    public static void errorExitingGame() {
        System.out.println("ERROR: an error occurred returning to main
Menu...exiting game...");
    }

    /**
     * Display 2 Cards in text form
     *
     * @param firstCard    the first card
     * @param secondCard   the second card
     * @param color        the color of the text
     * @param prefixString any required prefix string example 'Cards
Drawn:'
     */
    public static void displayTwoCards(Card firstCard, Card secondCard,
String color, String prefixString) {
        System.out.println(color + prefixString + " " + firstCard + " and "
+ secondCard + COLOR_WHITE);
    }

    /**
     * Display 3 Cards in text form
     *
     * @param firstCard    the first card
     * @param secondCard   the second card
     * @param thirdCard    the third card
     * @param color        the color of the text
     * @param prefixString any required prefix string example 'Cards
Drawn:'
     */
    public static void displayThreeCards(Card firstCard, Card secondCard,
Card thirdCard, String color, String prefixString) {
        System.out.print(color + prefixString + " " + firstCard + ", " +
secondCard + " and " + thirdCard + COLOR_WHITE);
    }

    /**
     * Display if the game is a stalemate and text prior to the last hand
     */
    public static void displayIsStalemate() {
        System.out.println(COLOR_RED + "\n \nGame is stalemate..\n" +
COLOR_WHITE);
    }

    /**
     * Method used to print out what round the user or computer failed at
     * @param currentRound round number of the last round
     */
    public static void failedAtRound(int currentRound){
        System.out.println(COLOR_RED + "Failed at Round: " +
(currentRound)+ " starting at zero, no valid selection possible...\n" +
COLOR_WHITE);
    }
    /**
     * Diplay win or lose output
     *
     * @param gameResult  the result of the game.
     * @param roundNumber the round number
     * @param isHuman     is the user human, eg not in demo mode/computer
plays mode.
     */
    public static void displayWinOrLoseOutPut(boolean gameResult, int
roundNumber, boolean isHuman) {
        if (gameResult) {
            if (isHuman) {
                System.out.println(COLOR_GREEN + "\nCongratz!! you have won
this Game! in " + (roundNumber) + " rounds starting at 0 because we are
programmers :)\n" + COLOR_WHITE);
            } else {
                System.out.println(COLOR_GREEN + "\nThe Computer has won
this game! in " + (roundNumber) + " rounds starting at 0 because we are
programmers :)\n" + COLOR_WHITE);
            }
        } else {
            if (isHuman) {
                System.out.println(COLOR_RED + "\nSadly you have lost this
Game, better luck next time!\n" + COLOR_WHITE);
            } else {
                System.out.println(COLOR_RED + "\nThe Computer has lost
this Game, oh no!\n" + COLOR_WHITE);
            }
        }
        System.out.println(COLOR_RED + "press enter to continue to the post
game menu..." + COLOR_WHITE);
    }

    /**
     * Print Returning to Game Menu...
     */
    public static void returningToGameMenu() {
        System.out.println("Returning to Game Menu...");
    }

    /**
     * Displays the lastGame as a replay
     *
     * @param lastGame last game to display in the replay
     */
    public static void displayActionReplayOfLastGame(Game lastGame) {
        Scanner keyPressScanner = new Scanner(System.in);
        System.out.println("\n------------------------------- Replay Round
Number: " + lastGame.getRoundQueue().getFront().getRoundNumber() + "-------
------------------------");

        //cards drawn this round.
        int drawn =
lastGame.getRoundQueue().getFront().getRoundMemoryDrawCards().countCards();
        System.out.println(Colors.COLOR_GREEN + "Number of Drawn cards that
round: " + drawn + ", cards drawn:" + Colors.COLOR_WHITE);

        //print the drawn cards from the rounds drawn card memory
        for (int i = 0; i < drawn; i++) {
            CardSlotsBag bag =
lastGame.getRoundQueue().getFront().getRoundMemoryDrawCards();
```



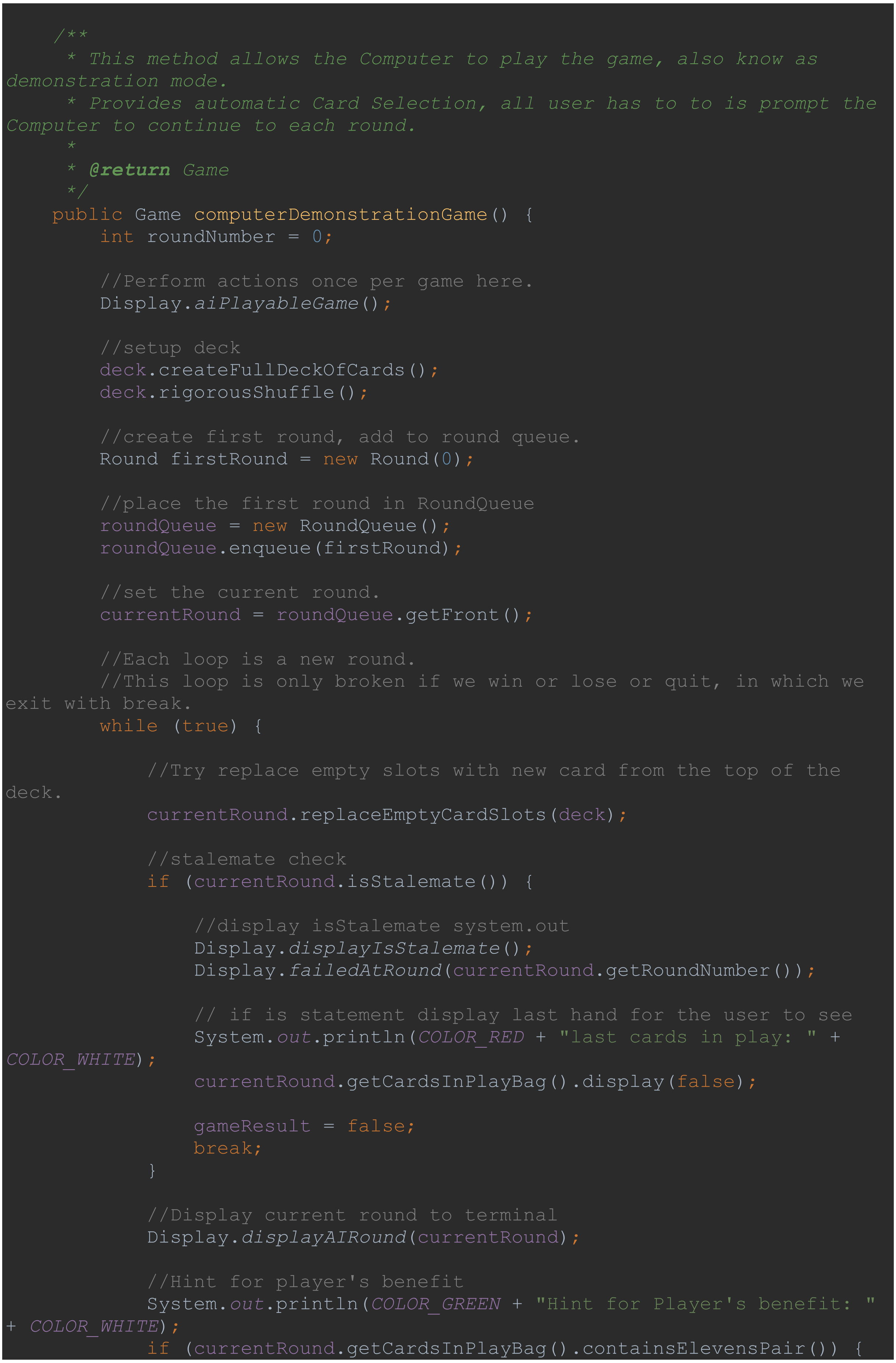
```
            String commaIfRequired = "";
            if (i == drawn - 1) {
                commaIfRequired = " ";
            } else {
                commaIfRequired = ", ";
            }
            System.out.print(Colors.COLOR_RED +
bag.cardAtPosition(i).toString() + commaIfRequired + Colors.COLOR_WHITE);
        }

        //print the discarded cards from the rounds discard card memory,
these cards are cards that where successfully removed.
        int discarded =
lastGame.getRoundQueue().getFront().getRoundMemoryDiscardCards().countCards
();
        System.out.println(Colors.COLOR_GREEN + "\nNumber of discarded
cards that round: " + discarded + ", discarded that round(successfully
removed): " + Colors.COLOR_WHITE);

        //Print out the discarded cards.
        for (int i = 0; i < discarded; i++) {
            CardSlotsBag bag1 =
lastGame.getRoundQueue().getFront().getRoundMemoryDiscardCards();
            String commaIfRequired = "";
            if (i == discarded - 1) {
                commaIfRequired = " ";
            } else {
                commaIfRequired = ", ";
            }
            System.out.print(Colors.COLOR_RED + " " +
bag1.cardAtPosition(i).toString() + commaIfRequired + Colors.COLOR_WHITE);
        }

        //State of Cards on table at the end of the round
        System.out.println(Colors.COLOR_GREEN + "\nState of Cards in play
at the end of the round, after discard cards where removed..." +
Colors.COLOR_WHITE);

lastGame.getRoundQueue().getFront().getCardsInPlayBag().display(false);

        //dequeue the round that's been displayed, as we no longer need it.
        lastGame.getRoundQueue().dequeue();

        //prompt and wait for input to go to the next round.
        System.out.println("\nPress any key to continue to the next replay
round...");
        keyPressScanner.nextLine();
    }

    /**
     * Ask user for input
     */
    public static void enterInput() {
        System.out.println();
        System.out.print(COLOR_GREEN + "select option > " + COLOR_WHITE);
    }

    /**
     * Tell user there input was invalid and they should try again, in
green color.
     */
    public static void invalidInput() {
        System.out.println();
        System.out.println(COLOR_RED + "Selected an Invalid Option....try
again." + COLOR_WHITE);
    }
}
```

Elevens.java

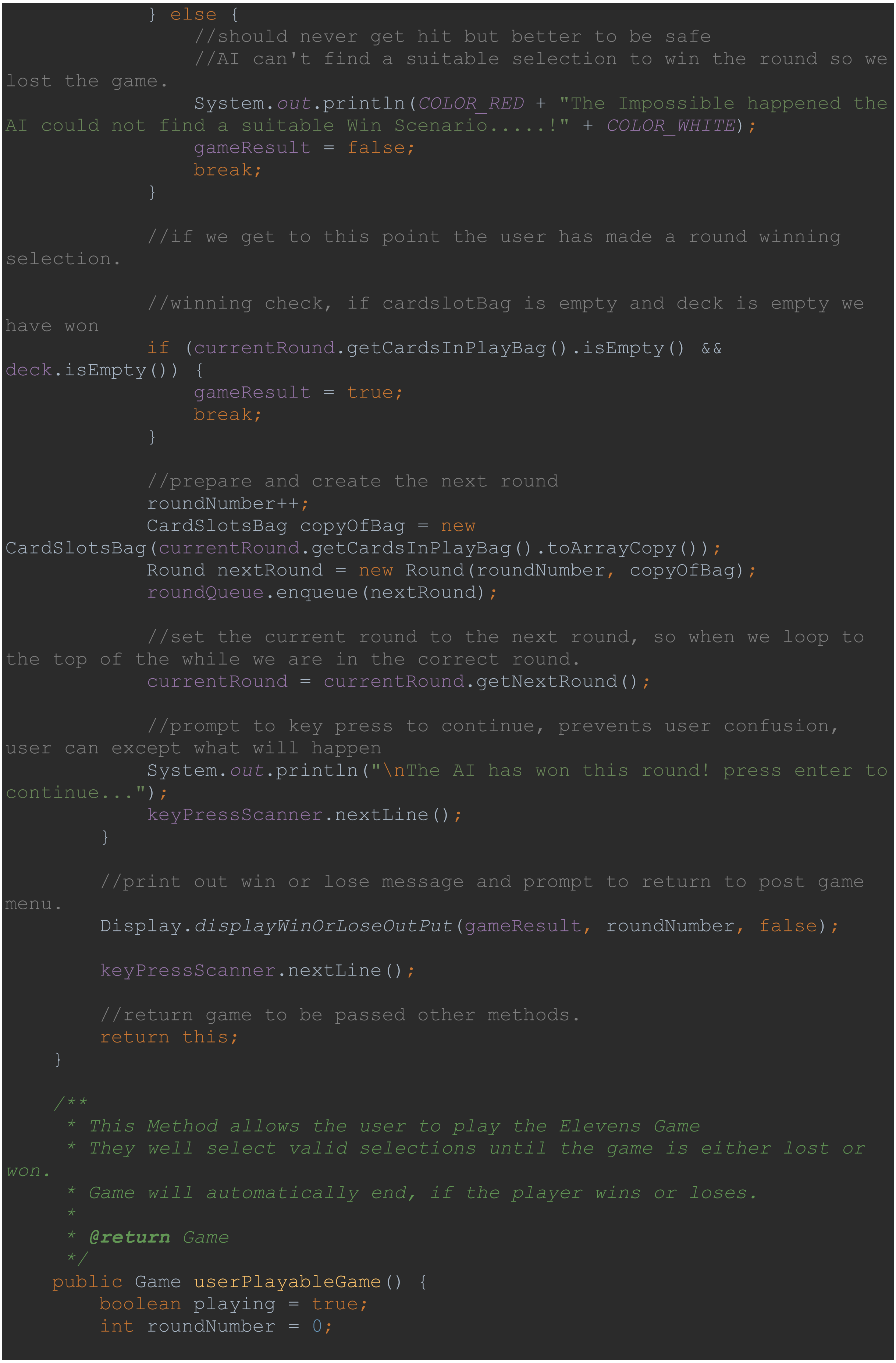
```java
package main;

/**
 * Class Elevens holds the main method
 */
public class Elevens {

    // Welcome the User, only once per application start.
    // Create Menu.
    private static void startElevensApplication() {
        Display.welcome();
        new Menu().MainMenu();
    }

    //Main method for the application
    public static void main(String[] args) {
        startElevensApplication();
    }
}
```

Game.java

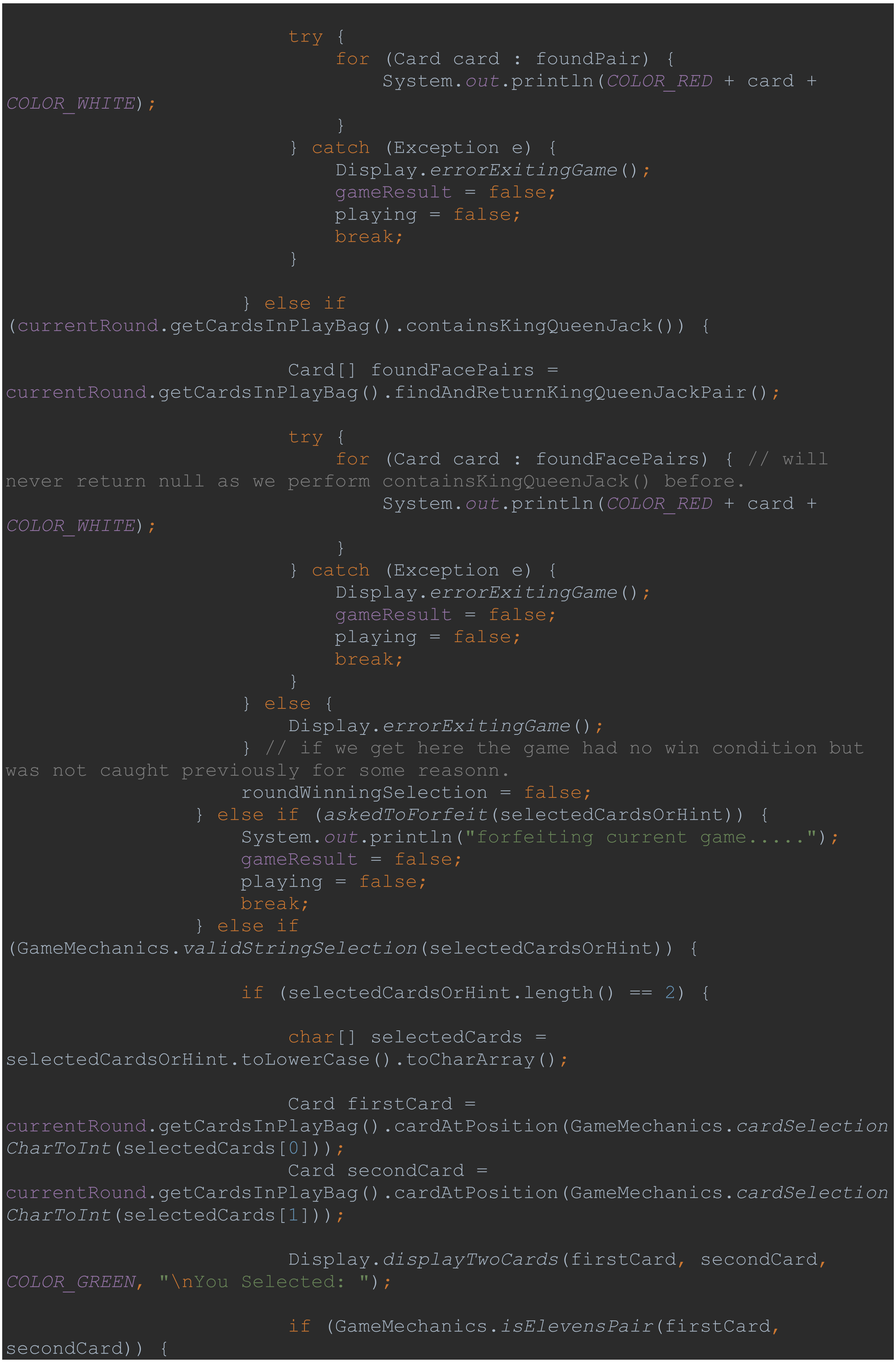
```java
package main;

import java.util.Scanner;

/**
 * This Class Represents a Game, holding all the required components to
play a game.
 * Including every round with memory of actions perform in the round.
 * The result of the game and the Deck and the discard deck.
 */
public class Game extends Colors {

    private Deck deck;
    private Deck discardDeck;
    private RoundQueue roundQueue;
    private Round currentRound;
    private Scanner scanner = new Scanner(System.in);
    private Scanner keyPressScanner = new Scanner(System.in);
    private boolean gameResult = false;

    /**
     * This constructor will consist all the components required to play a
game.
     */
    public Game() {
        this.deck = new Deck();
        this.discardDeck = new Deck();
        this.roundQueue = null;
    }

    /**
     * Checks if the input string equals 'hint'
     *
     * @param input the input string
     * @return boolean true if equals 'hint' or false if not
     */
    private static boolean askedForHint(String input) {
        return input.toLowerCase().equals("hint");
    }

    /**
     * Checks if the input string equals 'quit'
     *
     * @param input the input string
     * @return boolean true if equals 'quit' or false if not
     */
    private static boolean askedToForfeit(String input) {
        return input.toLowerCase().equals("quit");
    }

    /**
     * Get playable Deck
     *
     * @return Deck
     */
    public Deck getDeck() {
        return deck;
    }

    /**
     * Get the discard Deck, eg the deck of cards that where successfully
removed.
     *
     * @return the discard deck
     */
    public Deck getDiscardDeck() {
        return discardDeck;
    }

    /**
     * Return the Round Queue holding every Round.
     *
     * @return the round queue
     */
    public RoundQueue getRoundQueue() {
        return roundQueue;
    }

    /**
     * Get the current Round
     *
     * @return returns the current round
     */
    public Round getCurrentRound() {
        return currentRound;
    }

    /**
     * get the game result either win(true) or lose(false)
     *
     * @return boolean game result
     */
    public boolean getGameResult() {
        return gameResult;
    }
```



```
    /**
     * This method allows the Computer to play the game, also know as
demonstration mode.
     * Provides automatic Card Selection, all user has to to is prompt the
Computer to continue to each round.
     *
     * @return Game
     */
    public Game computerDemonstrationGame() {
        int roundNumber = 0;

        //Perform actions once per game here.
        Display.aiPlayableGame();

        //setup deck
        deck.createFullDeckOfCards();
        deck.rigorousShuffle();

        //create first round, add to round queue.
        Round firstRound = new Round(0);

        //place the first round in RoundQueue
        roundQueue = new RoundQueue();
        roundQueue.enqueue(firstRound);

        //set the current round.
        currentRound = roundQueue.getFront();

        //Each loop is a new round.
        //This loop is only broken if we win or lose or quit, in which we
exit with break.
        while (true) {

            //Try replace empty slots with new card from the top of the
deck.
            currentRound.replaceEmptyCardSlots(deck);

            //stalemate check
            if (currentRound.isStalemate()) {

                //display isStalemate system.out
                Display.displayIsStalemate();
                Display.failedAtRound(currentRound.getRoundNumber());

                // if is statement display last hand for the user to see
                System.out.println(COLOR_RED + "last cards in play: " +
COLOR_WHITE);
                currentRound.getCardsInPlayBag().display(false);

                gameResult = false;
                break;
            }

            //Display current round to terminal
            Display.displayAIRound(currentRound);

            //Hint for player's benefit
            System.out.println(COLOR_GREEN + "Hint for Player's benefit: "
+ COLOR_WHITE);
            if (currentRound.getCardsInPlayBag().containsElevensPair()) {
                Card[] foundPair =
currentRound.getCardsInPlayBag().findAndReturnElevensPair();
                try {
                    for (Card card : foundPair) {
                        System.out.println(COLOR_RED + " - " +card +
COLOR_WHITE);
                    }
                } catch (Exception e) {
                    Display.errorExitingGame();
                    gameResult = false;
                    break;
                }
            } else if
(currentRound.getCardsInPlayBag().containsKingQueenJack()) {
                Card[] foundFacePairs =
currentRound.getCardsInPlayBag().findAndReturnKingQueenJackPair();
                try {
                    for (Card card : foundFacePairs) { // will never return
null as we perform containsKingQueenJack();
                        System.out.println(COLOR_RED + card + COLOR_WHITE);
                    }
                } catch (Exception e) {
                    Display.errorExitingGame();
                    gameResult = false;
                    break;
                }
            }

            if (currentRound.getCardsInPlayBag().containsElevensPair()) {

                Card[] elevensPairArray =
currentRound.getCardsInPlayBag().findAndReturnElevensPair();

                if (elevensPairArray != null) {
                    System.out.println(COLOR_GREEN + "AI has selected
elevens pair:" + COLOR_WHITE);
                    for (Card card : elevensPairArray) {
                        System.out.println(" - " + card);

discardDeck.push(currentRound.getCardsInPlayBag().remove(card));
                        currentRound.updateDiscardCardMemory(card);
                    }
                    System.out.println();
                }

            } else if
(currentRound.getCardsInPlayBag().containsKingQueenJack()) {

                Card[] elevensFacePairsArray =
currentRound.getCardsInPlayBag().findAndReturnKingQueenJackPair();

                if (elevensFacePairsArray != null) {
                    System.out.println(COLOR_GREEN + "AI has selected face
card elevens pairs:" + COLOR_WHITE);
                    for (Card card : elevensFacePairsArray) {
                        System.out.print(" " + card);

discardDeck.push(currentRound.getCardsInPlayBag().remove(card));
                        currentRound.updateDiscardCardMemory(card);
                    }
                }
```



```
            } else {
                //should never get hit but better to be safe
                //AI can't find a suitable selection to win the round so we
lost the game.
                System.out.println(COLOR_RED + "The Impossible happened the
AI could not find a suitable Win Scenario.....!" + COLOR_WHITE);
                gameResult = false;
                break;
            }

            //if we get to this point the user has made a round winning
selection.

            //winning check, if cardslotBag is empty and deck is empty we
have won
            if (currentRound.getCardsInPlayBag().isEmpty() &&
deck.isEmpty()) {
                gameResult = true;
                break;
            }

            //prepare and create the next round
            roundNumber++;
            CardSlotsBag copyOfBag = new
CardSlotsBag(currentRound.getCardsInPlayBag().toArrayCopy());
            Round nextRound = new Round(roundNumber, copyOfBag);
            roundQueue.enqueue(nextRound);

            //set the current round to the next round, so when we loop to
the top of the while we are in the correct round.
            currentRound = currentRound.getNextRound();

            //prompt to key press to continue, prevents user confusion,
user can except what will happen
            System.out.println("\nThe AI has won this round! press enter to
continue...");
            keyPressScanner.nextLine();
        }

        //print out win or lose message and prompt to return to post game
menu.
        Display.displayWinOrLoseOutPut(gameResult, roundNumber, false);

        keyPressScanner.nextLine();

        //return game to be passed other methods.
        return this;
    }

    /**
     * This Method allows the user to play the Elevens Game
     * They well select valid selections until the game is either lost or
won.
     * Game will automatically end, if the player wins or loses.
     *
     * @return Game
     */
    public Game userPlayableGame() {
        boolean playing = true;
        int roundNumber = 0;
        //Perform actions once per game here.
        Display.userPlayableGame();

        //setup deck and shuffle
        deck.createFullDeckOfCards();
        deck.rigorousShuffle();

        //create first round, add to round queue.
        Round firstRound = new Round(0);

        //place the first round in RoundQueue
        roundQueue = new RoundQueue();
        roundQueue.enqueue(firstRound);

        //set the current round.
        currentRound = roundQueue.getFront();

        //Effectively each loop back to the top of the while(playing) is a
new round.
        while (playing) {

            //Try replace empty slots with new card from the top of the
deck.
            currentRound.replaceEmptyCardSlots(deck);

            //stalemate check
            if (currentRound.isStalemate()) {
                //display isStalemate system.out
                Display.displayIsStalemate();
                Display.failedAtRound(currentRound.getRoundNumber());
                currentRound.getCardsInPlayBag().display(true);
                gameResult = false;
                break;
            }

            //Display current round to terminal
            Display.displayRound(currentRound);

            //game is not a stalemate and we have not won, so allow user to
select cards.
            boolean roundWinningSelection = false;
            String selectedCardsOrHint = "";

            while (!roundWinningSelection) {

                System.out.println(COLOR_GREEN + "please select a valid
Elevens pair or pairs >" + COLOR_WHITE);

                selectedCardsOrHint = scanner.nextLine();

                //if they asked for a hint, workout a valid selection
                if (askedForHint(selectedCardsOrHint)) {

                    System.out.println(COLOR_GREEN + "Hint: " +
COLOR_WHITE);

                    if
(currentRound.getCardsInPlayBag().containsElevensPair()) {

                        Card[] foundPair =
currentRound.getCardsInPlayBag().findAndReturnElevensPair();
```



```
                        try {
                            for (Card card : foundPair) {
                                System.out.println(COLOR_RED + card +
COLOR_WHITE);
                            }
                        } catch (Exception e) {
                            Display.errorExitingGame();
                            gameResult = false;
                            playing = false;
                            break;
                        }

                    } else if
(currentRound.getCardsInPlayBag().containsKingQueenJack()) {

                        Card[] foundFacePairs =
currentRound.getCardsInPlayBag().findAndReturnKingQueenJackPair();

                        try {
                            for (Card card : foundFacePairs) { // will
never return null as we perform containsKingQueenJack() before.
                                System.out.println(COLOR_RED + card +
COLOR_WHITE);
                            }
                        } catch (Exception e) {
                            Display.errorExitingGame();
                            gameResult = false;
                            playing = false;
                            break;
                        }
                    } else {
                        Display.errorExitingGame();
                    } // if we get here the game had no win condition but
was not caught previously for some reasonn.
                    roundWinningSelection = false;
                } else if (askedToForfeit(selectedCardsOrHint)) {
                    System.out.println("forfeiting current game.....");
                    gameResult = false;
                    playing = false;
                    break;
                } else if
(GameMechanics.validStringSelection(selectedCardsOrHint)) {

                    if (selectedCardsOrHint.length() == 2) {

                        char[] selectedCards =
selectedCardsOrHint.toLowerCase().toCharArray();

                        Card firstCard =
currentRound.getCardsInPlayBag().cardAtPosition(GameMechanics.cardSelection
CharToInt(selectedCards[0]));
                        Card secondCard =
currentRound.getCardsInPlayBag().cardAtPosition(GameMechanics.cardSelection
CharToInt(selectedCards[1]));

                        Display.displayTwoCards(firstCard, secondCard,
COLOR_GREEN, "\nYou Selected: ");

                        if (GameMechanics.isElevensPair(firstCard,
secondCard)) {
```

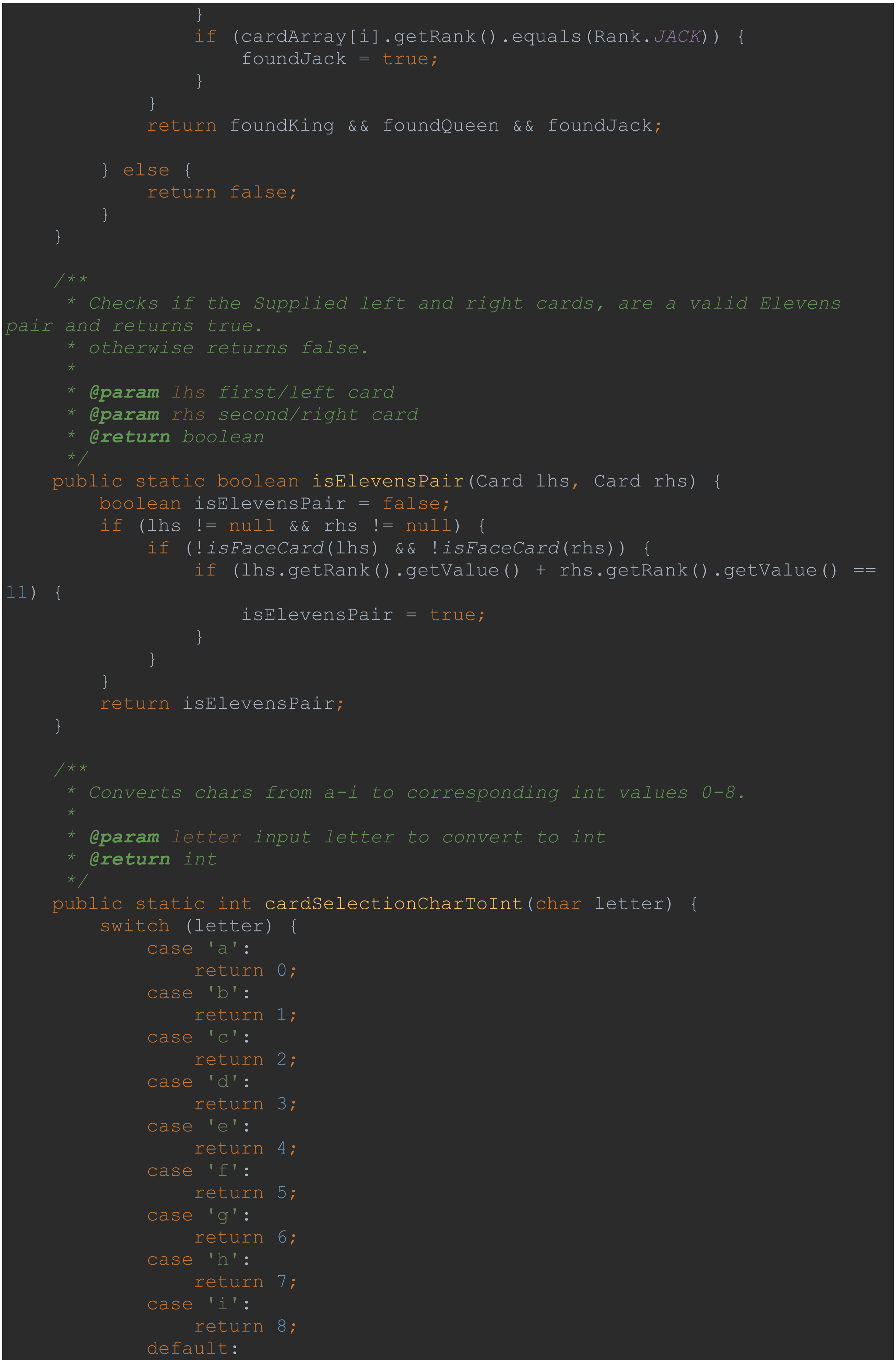
```
                            //Valid selection we can now remove cards and
move to next round
                            Display.displayTwoCards(firstCard, secondCard,
Colors.COLOR_GREEN, "\nValid Selection! Your selected cards were a valid
Elevens pair: ");

                            //remove the valid cards.

discardDeck.push(currentRound.getCardsInPlayBag().remove(firstCard));

discardDeck.push(currentRound.getCardsInPlayBag().remove(secondCard));

                            //update round memory for replay feature

currentRound.updateDiscardCardMemory(firstCard);

currentRound.updateDiscardCardMemory(secondCard);

                            roundWinningSelection = true;
                        } else {
                            //invalid selection, prompt to try again
                            Display.displayTwoCards(firstCard, secondCard,
Colors.COLOR_RED, "\nInvalid Selection: Your select cards were not a valid
Elevens pair... ");
                            roundWinningSelection = false;
                        }

                    } else if (selectedCardsOrHint.length() == 3) {
                        char[] selectedCards =
selectedCardsOrHint.toLowerCase().toCharArray();

                        Card firstCard =
currentRound.getCardsInPlayBag().cardAtPosition(GameMechanics.cardSelection
CharToInt(selectedCards[0]));
                        Card secondCard =
currentRound.getCardsInPlayBag().cardAtPosition(GameMechanics.cardSelection
CharToInt(selectedCards[1]));
                        Card thirdCard =
currentRound.getCardsInPlayBag().cardAtPosition(GameMechanics.cardSelection
CharToInt(selectedCards[2]));

                        Display.displayThreeCards(firstCard, secondCard,
thirdCard, Colors.COLOR_GREEN, "\nYou Selected 3 face cards: ");

                        if (GameMechanics.isFacePairs(firstCard,
secondCard, thirdCard)) {

                            //Valid selection we can now remove cards and
move to next round
                            Display.displayThreeCards(firstCard,
secondCard, thirdCard, Colors.COLOR_GREEN, "\nValid Selection! Your
selected cards contained a King, Queen and a Jack...");

                            //remove the valid cards.

discardDeck.push(currentRound.getCardsInPlayBag().remove(firstCard));

discardDeck.push(currentRound.getCardsInPlayBag().remove(secondCard));

discardDeck.push(currentRound.getCardsInPlayBag().remove(thirdCard));
```

```
                            //update round memory for replay feature

currentRound.updateDiscardCardMemory(firstCard);

currentRound.updateDiscardCardMemory(secondCard);

currentRound.updateDiscardCardMemory(thirdCard);

                            roundWinningSelection = true;
                        } else {
                            //invalid selection, prompt to try again
                            Display.displayThreeCards(firstCard,
secondCard, thirdCard, Colors.COLOR_RED, "\nInvalid Selection: Your select
cards did not contain a King, Queen and Jack... ");
                            System.out.println(firstCard + ", " +
secondCard + ", " + thirdCard);
                            roundWinningSelection = false;
                        }
                    }
                }
            }

            //winning check, if cardslotBag is empty and deck is empty we
have won
            if (currentRound.getCardsInPlayBag().isEmpty() &&
deck.isEmpty()) {
                gameResult = true;
                break;
            }

            //if we get to this point the user has made a round winning
selection.
            //prepare and create the next round
            roundNumber++;
            CardSlotsBag copyOfBag = new
CardSlotsBag(currentRound.getCardsInPlayBag().toArrayCopy());
            Round nextRound = new Round(roundNumber, copyOfBag);
            roundQueue.enqueue(nextRound);

            //set the current round to the next round, so when we loop to
the top of the while we are in the correct round.
            currentRound = currentRound.getNextRound();

            //prompt to key press to continue, prevents user confusion,
user can except what will happen
            if (playing) {
                System.out.println("\nYou have Won this round! press enter
to continue...");
            }
            keyPressScanner.nextLine();
        }

        //print out win or lose message and prompt to return to post game
menu.
        Display.displayWinOrLoseOutPut(gameResult, roundNumber, true);

        //wait for key press
        keyPressScanner.nextLine();

        return this;
```

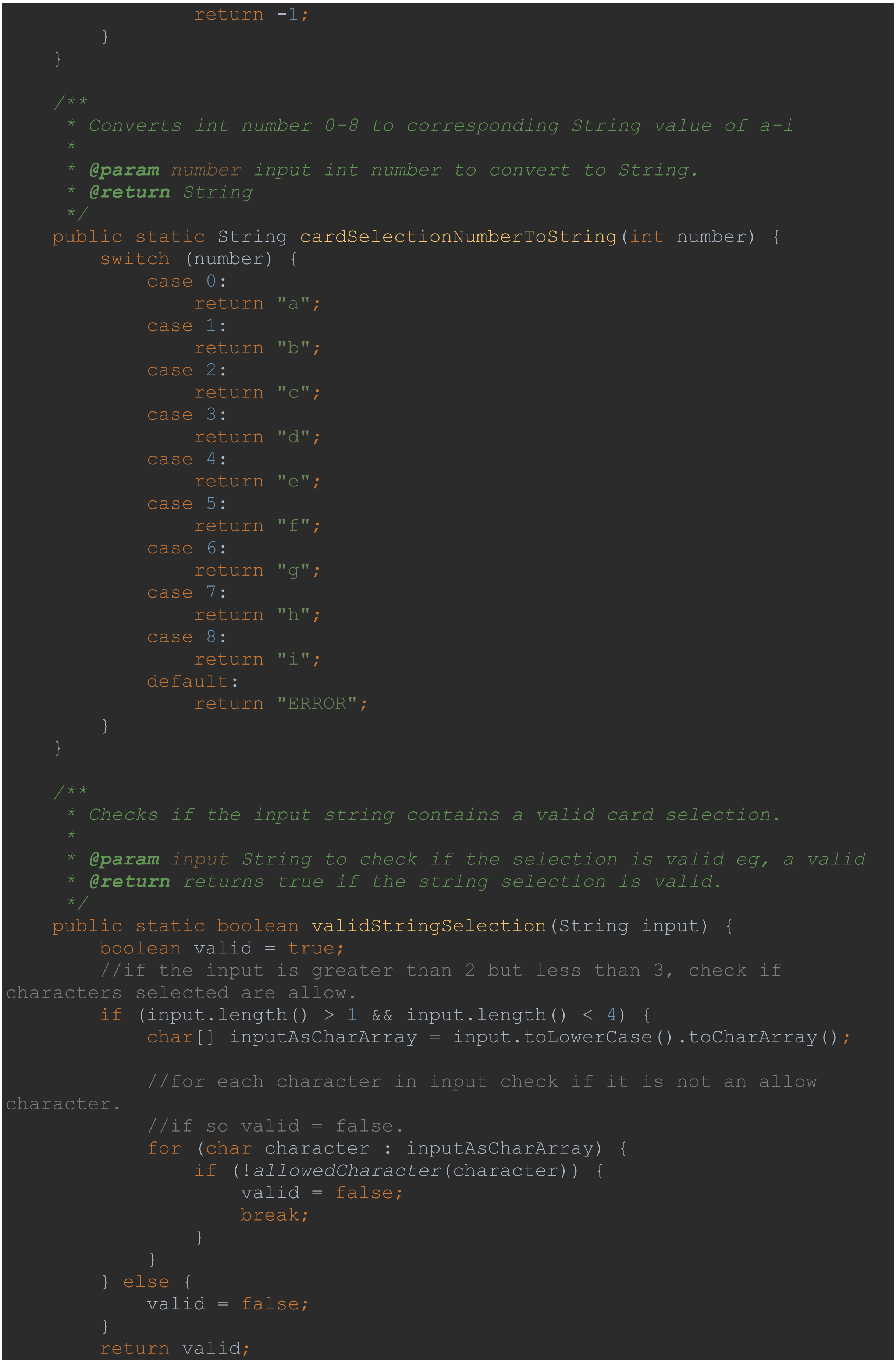
```
    }
}
```

GameMechanics.java

```
package main;

public class GameMechanics {

    /**
     * Checks if a Card is a face card
     *
     * @param aCard the card in question.
     * @return boolean
     */
    public static boolean isFaceCard(Card aCard) {
        //make sure the card is not null
        if (aCard != null) {

            //if not null look for a Face Card rank.
            if (aCard.getRank().equals(Rank.KING)
                    || aCard.getRank().equals(Rank.QUEEN)
                    || aCard.getRank().equals(Rank.JACK)) {
                return true;
            } else {
                return false;
            }
        } else {
            return false;
        }
    }

    /**
     * Checks if the supplied Cards are face pairs, eg one of each King,
Queen and Jack.
     * Otherwise returns false.
     *
     * @param oneCard   the first card
     * @param twoCard   the second card
     * @param threeCard the third card
     * @return boolean
     */
    public static boolean isFacePairs(Card oneCard, Card twoCard, Card
threeCard) {
        if (isFaceCard(oneCard) && isFaceCard(twoCard) &&
isFaceCard(threeCard)) {
            boolean foundKing = false;
            boolean foundQueen = false;
            boolean foundJack = false;

            Card[] cardArray = {oneCard, twoCard, threeCard};

            for (int i = 0; i < cardArray.length; i++) {
                if (cardArray[i].getRank().equals(Rank.KING)) {
                    foundKing = true;
                }
                if (cardArray[i].getRank().equals(Rank.QUEEN)) {
                    foundQueen = true;
```



```
                }
                if (cardArray[i].getRank().equals(Rank.JACK)) {
                    foundJack = true;
                }
            }
            return foundKing && foundQueen && foundJack;

        } else {
            return false;
        }
    }

    /**
     * Checks if the Supplied left and right cards, are a valid Elevens
pair and returns true.
     * otherwise returns false.
     *
     * @param lhs first/left card
     * @param rhs second/right card
     * @return boolean
     */
    public static boolean isElevensPair(Card lhs, Card rhs) {
        boolean isElevensPair = false;
        if (lhs != null && rhs != null) {
            if (!isFaceCard(lhs) && !isFaceCard(rhs)) {
                if (lhs.getRank().getValue() + rhs.getRank().getValue() ==
11) {
                    isElevensPair = true;
                }
            }
        }
        return isElevensPair;
    }

    /**
     * Converts chars from a-i to corresponding int values 0-8.
     *
     * @param letter input letter to convert to int
     * @return int
     */
    public static int cardSelectionCharToInt(char letter) {
        switch (letter) {
            case 'a':
                return 0;
            case 'b':
                return 1;
            case 'c':
                return 2;
            case 'd':
                return 3;
            case 'e':
                return 4;
            case 'f':
                return 5;
            case 'g':
                return 6;
            case 'h':
                return 7;
            case 'i':
                return 8;
            default:
```



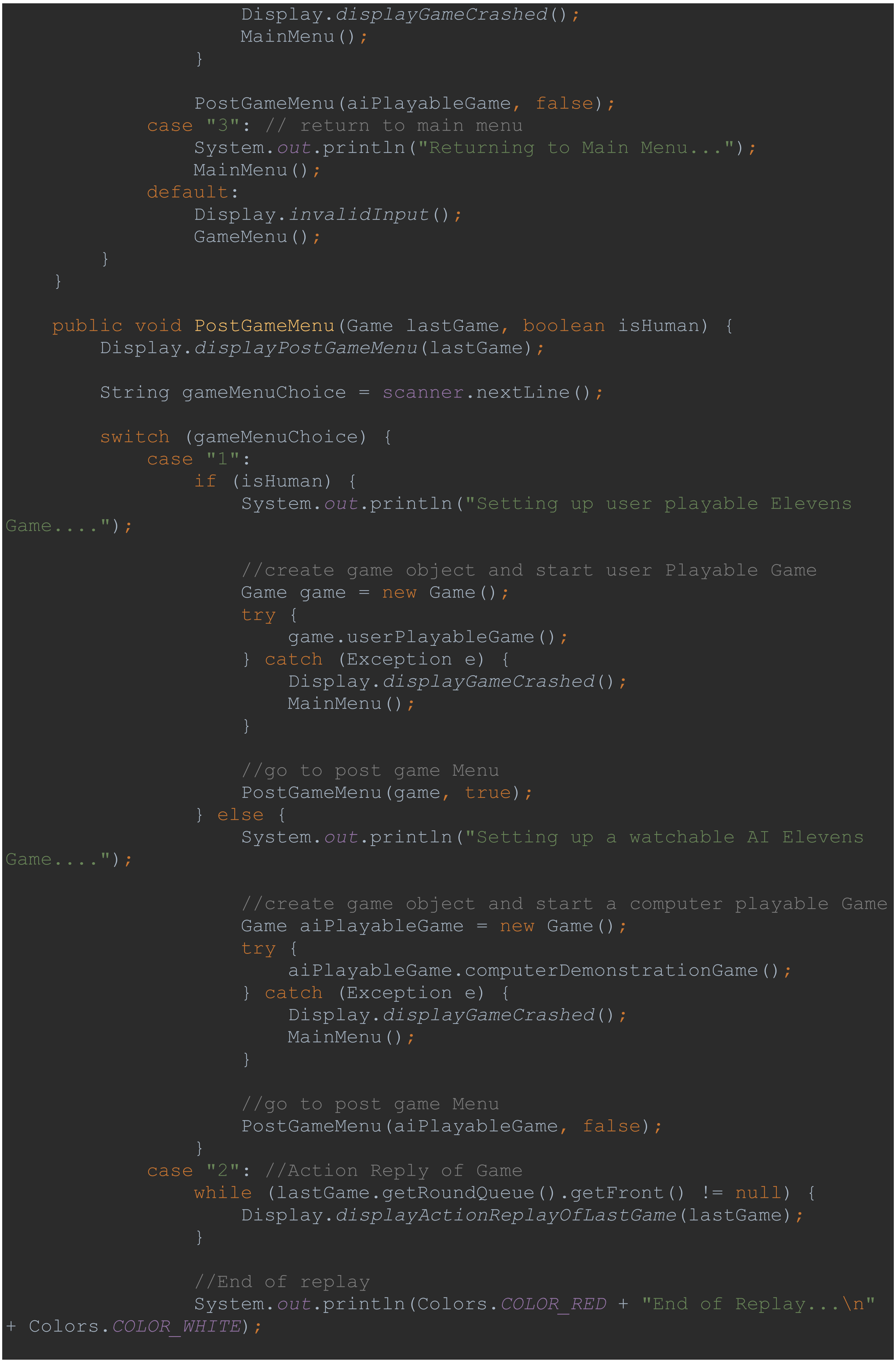
```
                return -1;
        }
    }

    /**
     * Converts int number 0-8 to corresponding String value of a-i
     *
     * @param number input int number to convert to String.
     * @return String
     */
    public static String cardSelectionNumberToString(int number) {
        switch (number) {
            case 0:
                return "a";
            case 1:
                return "b";
            case 2:
                return "c";
            case 3:
                return "d";
            case 4:
                return "e";
            case 5:
                return "f";
            case 6:
                return "g";
            case 7:
                return "h";
            case 8:
                return "i";
            default:
                return "ERROR";
        }
    }

    /**
     * Checks if the input string contains a valid card selection.
     *
     * @param input String to check if the selection is valid eg, a valid
     * @return returns true if the string selection is valid.
     */
    public static boolean validStringSelection(String input) {
        boolean valid = true;
        //if the input is greater than 2 but less than 3, check if 
characters selected are allow.
        if (input.length() > 1 && input.length() < 4) {
            char[] inputAsCharArray = input.toLowerCase().toCharArray();

            //for each character in input check if it is not an allow 
character.
            //if so valid = false.
            for (char character : inputAsCharArray) {
                if (!allowedCharacter(character)) {
                    valid = false;
                    break;
                }
            }
        } else {
            valid = false;
        }
        return valid;
    }

    /**
     * Checks if the selected char is a valid card that can be choosen.
     * Does not check if the card slot has a card.
     *
     * @param letter char to check if is one of the slots.
     * @return returns true if the selected char is one of a-i
     */
    public static boolean allowedCharacter(char letter) {
        char[] allowedChars = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
'i'};
        boolean contains = false;

        for (char character : allowedChars) {
            if (character == letter) {
                contains = true;
                break;
            }
        }
        return contains;
    }
}
```

House.java

```
package main;

/**
 * Enum representing, each house or suite a card can be in.
 */
public enum House {
    HEARTS("Hearts"),
    DIAMONDS("Diamonds"),
    SPADES("Spades"),
    CLUBS("Clubs");

    private String houseName;

    /**
     * Constructor for house.
     *
     * @param house one of the houses of a suite of cards.
     */
    House(String house) {
        this.houseName = house;
    }

    /**
     * Return the String value of the House.
     *
     * @return String
     */
    public String toString() {
        return houseName;
    }
}
```

Menu.java

```
package main;

import java.util.Scanner;

/**
 * A class that represents each menu, each menu then creates game objects
or other objects whenn the user selects
 * and option.
 */
public class Menu {

    private Scanner scanner = new Scanner(System.in);

    //only used for when user is prompted to key press.
    private Scanner keyPressScanner = new Scanner(System.in);

    public void MainMenu() {
        Display.mainMenu();
        String mainMenuChoice = scanner.nextLine();
        switch (mainMenuChoice) {
            case "1": // To Game Menu
                GameMenu();
                MainMenu();
            case "2": // Exit to desktop
                System.out.println("Exiting Application....Goodbye!!");
                System.exit(0);
            default:
                Display.invalidInput();
                MainMenu();
        }
    }

    public void GameMenu() {
        Display.gameMenu();

        String gameMenuChoice = scanner.nextLine();

        switch (gameMenuChoice) {
            case "1": // Setup user playable Elevens main.Game
                System.out.println("Setting up user playable Elevens
Game....");

                Game game = new Game();
                try {
                    game.userPlayableGame();
                } catch (Exception e) {
                    Display.displayGameCrashed();
                    MainMenu();
                }

                PostGameMenu(game, true);
            case "2": // AI playable Elevens game
                System.out.println("Setting up a watchable AI Elevens
Game....");

                Game aiPlayableGame = new Game();
                try {
                    aiPlayableGame.computerDemonstrationGame();
                } catch (Exception e) {
```



```
                    Display.displayGameCrashed();
                    MainMenu();
                }

                PostGameMenu(aiPlayableGame, false);
            case "3": // return to main menu
                System.out.println("Returning to Main Menu...");
                MainMenu();
            default:
                Display.invalidInput();
                GameMenu();
        }
    }

    public void PostGameMenu(Game lastGame, boolean isHuman) {
        Display.displayPostGameMenu(lastGame);

        String gameMenuChoice = scanner.nextLine();

        switch (gameMenuChoice) {
            case "1":
                if (isHuman) {
                    System.out.println("Setting up user playable Elevens
Game....");

                    //create game object and start user Playable Game
                    Game game = new Game();
                    try {
                        game.userPlayableGame();
                    } catch (Exception e) {
                        Display.displayGameCrashed();
                        MainMenu();
                    }

                    //go to post game Menu
                    PostGameMenu(game, true);
                } else {
                    System.out.println("Setting up a watchable AI Elevens
Game....");

                    //create game object and start a computer playable Game
                    Game aiPlayableGame = new Game();
                    try {
                        aiPlayableGame.computerDemonstrationGame();
                    } catch (Exception e) {
                        Display.displayGameCrashed();
                        MainMenu();
                    }

                    //go to post game Menu
                    PostGameMenu(aiPlayableGame, false);
                }
            case "2": //Action Reply of Game
                while (lastGame.getRoundQueue().getFront() != null) {
                    Display.displayActionReplayOfLastGame(lastGame);
                }

                //End of replay
                System.out.println(Colors.COLOR_RED + "End of Replay...\n"
+ Colors.COLOR_WHITE);
```

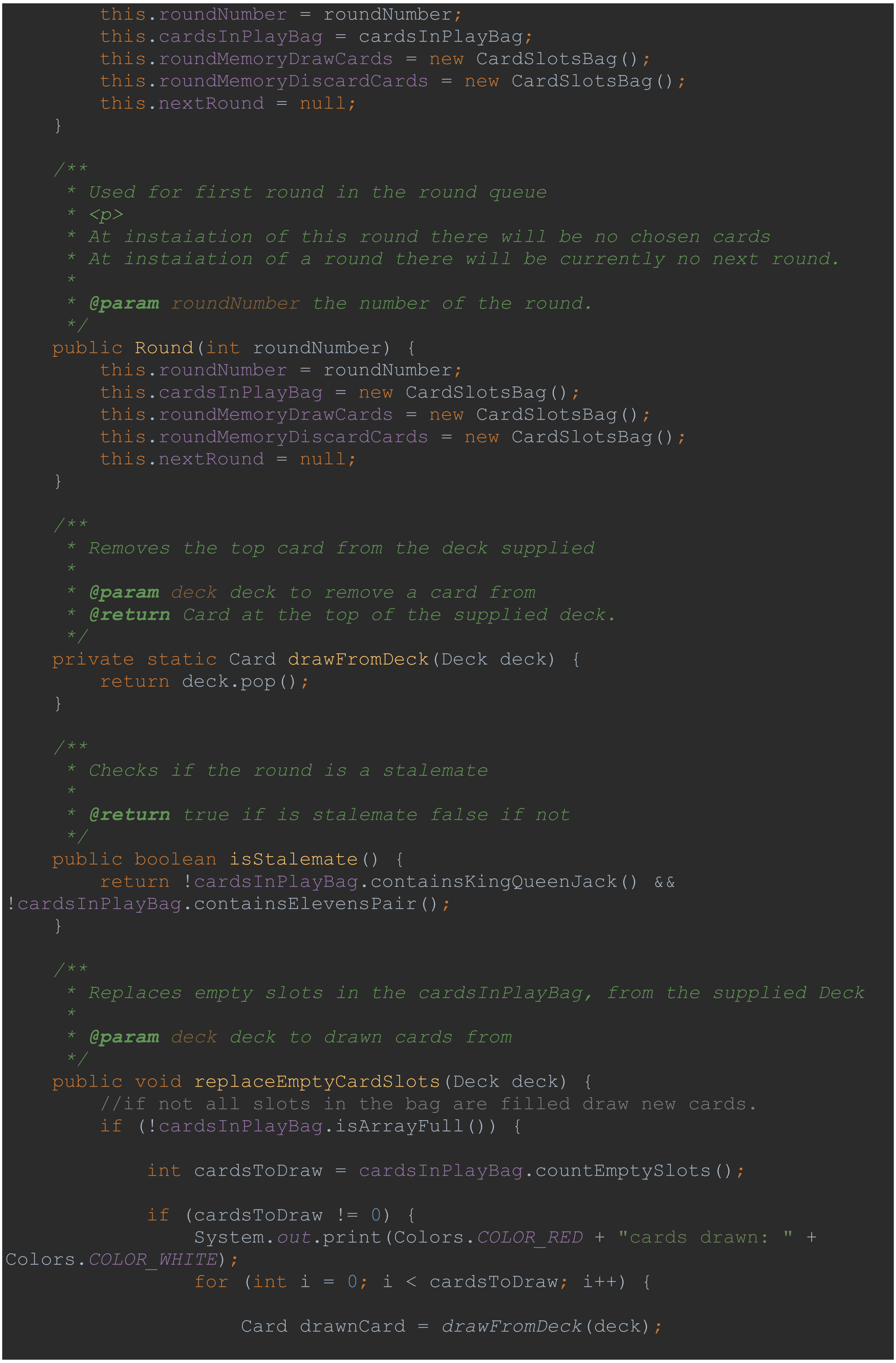
```
                //Wait for input
                keyPressScanner.nextLine();
                Display.returningToGameMenu();
                GameMenu();
            case "3": //Return to main.Game main.Menu
                Display.returningToGameMenu();
                GameMenu();
            default: //Notify Invalid input and re-display menu
                Display.invalidInput();
                PostGameMenu(lastGame, isHuman);
        }
    }
}
```

Rank.java

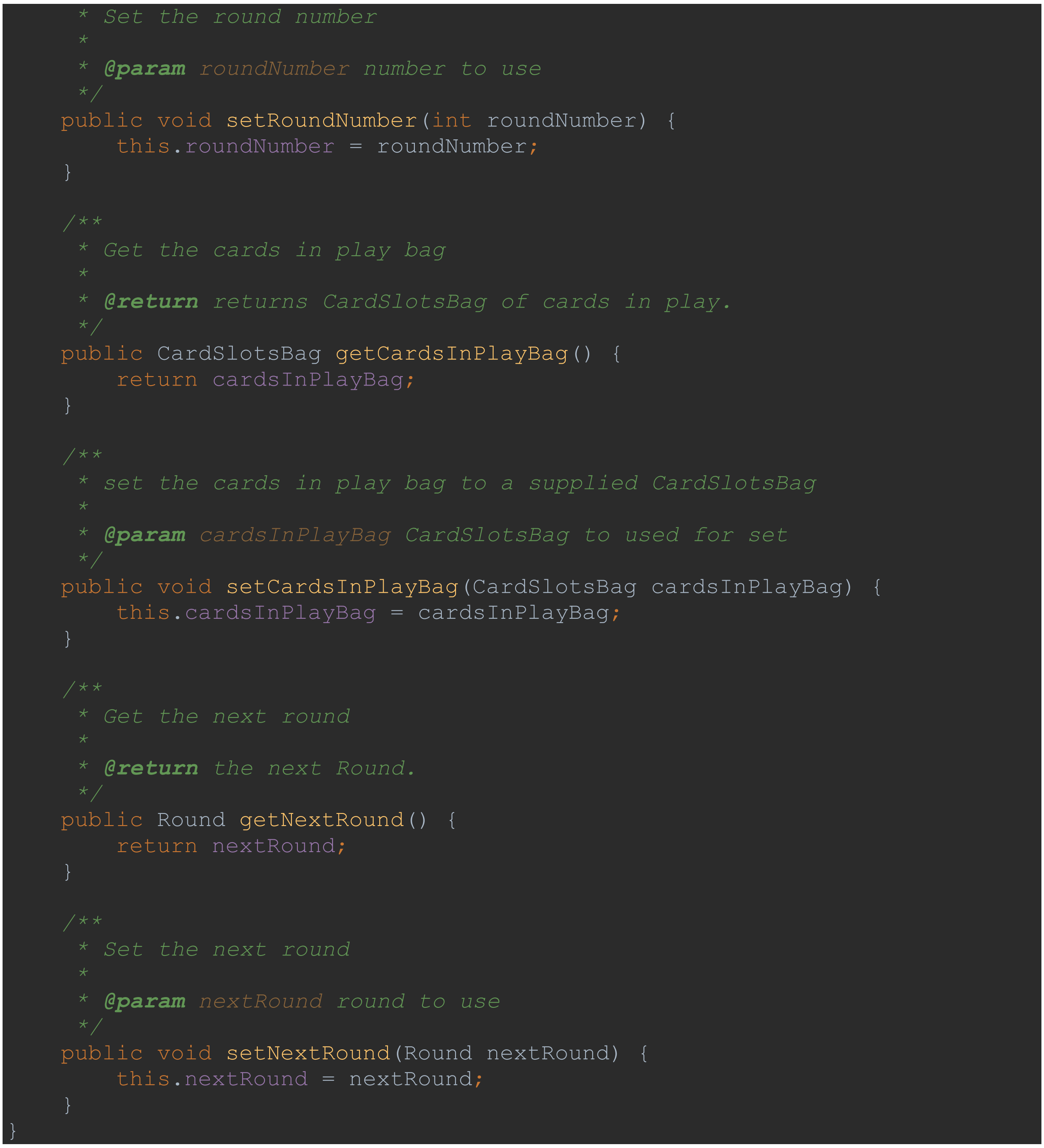
```
package main;

/**
 * Note face cards values are set to -1, as in Elevens face cards to not have a usable value.
 * So we have assigned face cards a value of -1.
 */
public enum Rank {
    KING("King", -1),
    QUEEN("Queen", -1),
    JACK("Jack", -1),
    ACE("Ace", 1),
    TWO("Two", 2),
    THREE("Three", 3),
    FOUR("Four", 4),
    FIVE("Five", 5),
    SIX("Six", 6),
    SEVEN("Seven", 7),
    EIGHT("Eight", 8),
    NINE("Nine", 9),
    TEN("Ten", 10);

    private String rank;
    private int value;

    /**
     * Constructor to create a main.Rank Object,
     * Only Require a constructor with all parameters.
     * Object fields will never be defaulted for this class.
     *
     * @param rank  rank of the card example King or Ace or One.
     * @param value the integer value of the Card.
     */
    Rank(String rank, int value) {
        this.rank = rank;
        this.value = value;
    }

    /***
     * Get the main.Rank in Sting format.
     * @return String
     */
    public String getRank() {
        return rank;
    }

    /***
     * Get the integer value of a card.
     * @return Int
     */
    public int getValue() {
        return value;
    }

    /**
     * Return the String value of the main.Rank.
     * Override to string for easy use in Strings.
     *
     * @return String
     */
    @Override
    public String toString() {
        return rank;
    }
}
```

Round.java

```
package main;

/**
 * This Class represents each round within a game,
 * It stores information about each round.
 * Such as the card in play in a round and memory of events
 */
public class Round {

    private int roundNumber;
    private Round nextRound;

    //Cards in play in current round.
    private CardSlotsBag cardsInPlayBag;

    //Used to remember each rounds events, such as drawn cards and
discarded cards.
    private CardSlotsBag roundMemoryDrawCards;
    private CardSlotsBag roundMemoryDiscardCards;

    /**
     * Used for subsequent rounds
     * <p>
     * cardSlots will be filled with the cardSlots of the previous round.
     * <p>
     * At instaiation of a round there will be no chosen cards
     * At instaiation of a round there will be currently no next round.
     *
     * @param roundNumber    the number of the round.
     * @param cardsInPlayBag a bag for Cards representing cards in play.
     */
    public Round(int roundNumber, CardSlotsBag cardsInPlayBag) {
```



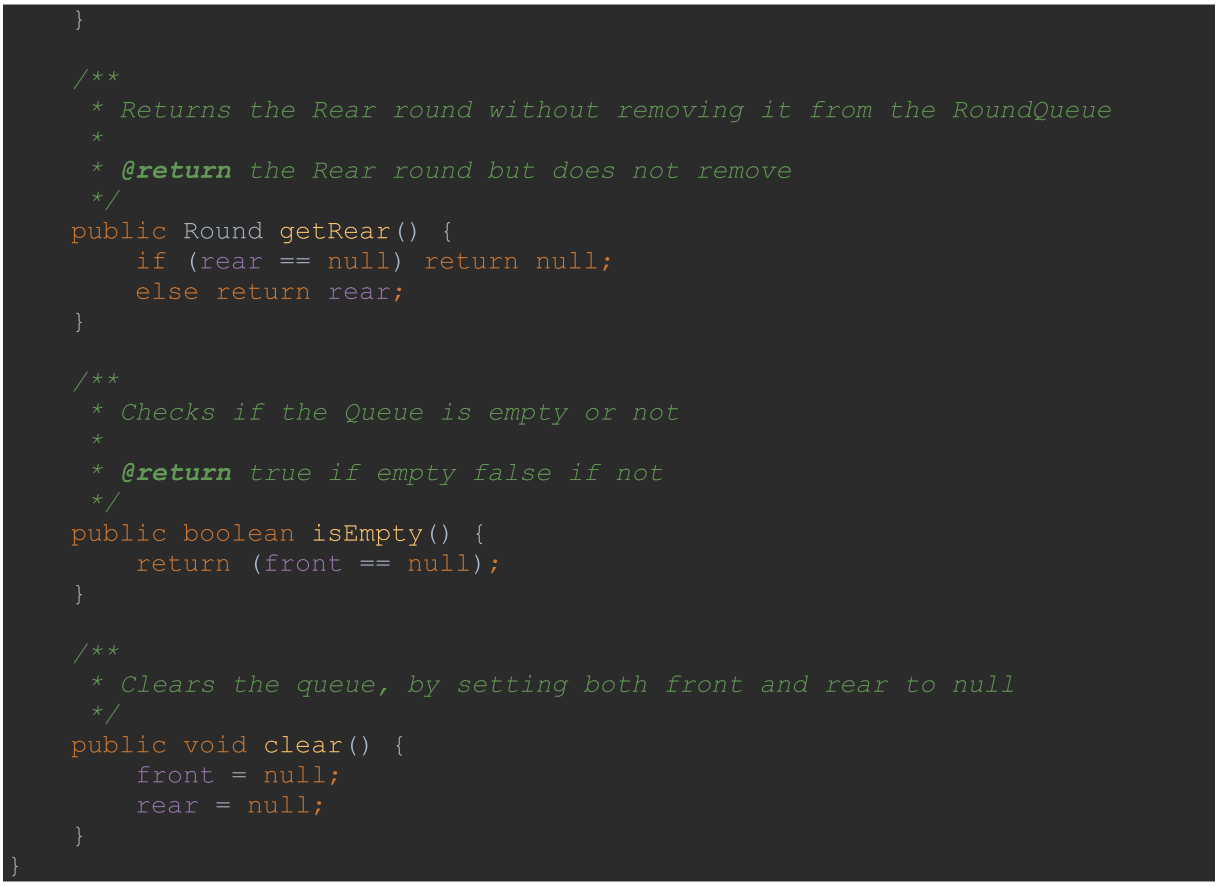
```
        this.roundNumber = roundNumber;
        this.cardsInPlayBag = cardsInPlayBag;
        this.roundMemoryDrawCards = new CardSlotsBag();
        this.roundMemoryDiscardCards = new CardSlotsBag();
        this.nextRound = null;
    }

    /**
     * Used for first round in the round queue
     * <p>
     * At instaiation of this round there will be no chosen cards
     * At instaiation of a round there will be currently no next round.
     *
     * @param roundNumber the number of the round.
     */
    public Round(int roundNumber) {
        this.roundNumber = roundNumber;
        this.cardsInPlayBag = new CardSlotsBag();
        this.roundMemoryDrawCards = new CardSlotsBag();
        this.roundMemoryDiscardCards = new CardSlotsBag();
        this.nextRound = null;
    }

    /**
     * Removes the top card from the deck supplied
     *
     * @param deck deck to remove a card from
     * @return Card at the top of the supplied deck.
     */
    private static Card drawFromDeck(Deck deck) {
        return deck.pop();
    }

    /**
     * Checks if the round is a stalemate
     *
     * @return true if is stalemate false if not
     */
    public boolean isStalemate() {
        return !cardsInPlayBag.containsKingQueenJack() &&
!cardsInPlayBag.containsElevensPair();
    }

    /**
     * Replaces empty slots in the cardsInPlayBag, from the supplied Deck
     *
     * @param deck deck to drawn cards from
     */
    public void replaceEmptyCardSlots(Deck deck) {
        //if not all slots in the bag are filled draw new cards.
        if (!cardsInPlayBag.isArrayFull()) {

            int cardsToDraw = cardsInPlayBag.countEmptySlots();

            if (cardsToDraw != 0) {
                System.out.print(Colors.COLOR_RED + "cards drawn: " +
Colors.COLOR_WHITE);
                for (int i = 0; i < cardsToDraw; i++) {

                    Card drawnCard = drawFromDeck(deck);
                    String postFixComma = ", ";
                    //Make sure drawnCard is not null, happens when deck is
empty.
                    if (drawnCard != null) {

                        //remove comma on last card drawn
                        if(i == cardsToDraw -1) postFixComma = "";

                        System.out.print(" " + drawnCard.toString() +
postFixComma);

                        cardsInPlayBag.addNewEntry(drawnCard);

                        //Add to round memory of drawn Cards for replay
feature
                        roundMemoryDrawCards.addNewEntry(drawnCard);
                    }
                }
            }
        } else {
            System.out.println("card slots are full no cards drawn...");
        }
    }

    /**
     * Get the memory of the cards drawn in this round
     *
     * @return a CardSlotsBag off cards drawn
     */
    public CardSlotsBag getRoundMemoryDrawCards() {
        return roundMemoryDrawCards;
    }

    /**
     * Get the memory of the cards discarded in the currrent round.
     *
     * @return a CardSlotsBag off cards discarded
     */
    public CardSlotsBag getRoundMemoryDiscardCards() {
        return roundMemoryDiscardCards;
    }

    /**
     * Update the discarded card memory,
     *
     * @param card the card to add
     */
    public void updateDiscardCardMemory(Card card) {
        this.roundMemoryDiscardCards.addNewEntry(card);
    }

    /**
     * Get the round number
     *
     * @return int number of the round
     */
    public int getRoundNumber() {
        return roundNumber;
    }

    /**
```



```java
     * Set the round number
     *
     * @param roundNumber number to use
     */
    public void setRoundNumber(int roundNumber) {
        this.roundNumber = roundNumber;
    }

    /**
     * Get the cards in play bag
     *
     * @return returns CardSlotsBag of cards in play.
     */
    public CardSlotsBag getCardsInPlayBag() {
        return cardsInPlayBag;
    }

    /**
     * set the cards in play bag to a supplied CardSlotsBag
     *
     * @param cardsInPlayBag CardSlotsBag to used for set
     */
    public void setCardsInPlayBag(CardSlotsBag cardsInPlayBag) {
        this.cardsInPlayBag = cardsInPlayBag;
    }

    /**
     * Get the next round
     *
     * @return the next Round.
     */
    public Round getNextRound() {
        return nextRound;
    }

    /**
     * Set the next round
     *
     * @param nextRound round to use
     */
    public void setNextRound(Round nextRound) {
        this.nextRound = nextRound;
    }
}
```

RoundQueue.java

```java
package main;

import Interfaces.QueueInterface;

/**
 * This Class is a object that holds information about each round.
 * As new rounds are created they will be enqueued and when we replay a
round we can dequeue the round
 */
public class RoundQueue implements QueueInterface<Round> {

    private Round front, rear;

    /**
     * Constructor for a RoundQueue, creates an queue with no front or rear
     */
    public RoundQueue() {
        front = null;
        rear = null;
    }

    /**
     * Enqueue a new round
     *
     * @param newRound the round to enqueue
     */
    public void enqueue(Round newRound) {
        if (front == null) {
            front = newRound;
            rear = newRound;
        } else {
            rear.setNextRound(newRound);
            rear = newRound;
        }
    }

    /**
     * Removes the first round in the queue.
     *
     * @return the removed Round
     */
    public Round dequeue() {
        if (front == null) return null;
        else {
            Round valueToReturn = front;
            front = front.getNextRound();
            if (front == null) rear = null;
            return valueToReturn;
        }
    }

    /**
     * Returns the front round without removing it from the RoundQueue
     *
     * @return the Front round but does not remove
     */
    public Round getFront() {
        if (front == null) return null;
        else return front;
```



```java
    }

    /**
     * Returns the Rear round without removing it from the RoundQueue
     *
     * @return the Rear round but does not remove
     */
    public Round getRear() {
        if (rear == null) return null;
        else return rear;
    }

    /**
     * Checks if the Queue is empty or not
     *
     * @return true if empty false if not
     */
    public boolean isEmpty() {
        return (front == null);
    }

    /**
     * Clears the queue, by setting both front and rear to null
     */
    public void clear() {
        front = null;
        rear = null;
    }
}
```