```java
import java.util.Objects;
import java.util.Random;

public class Card {
    private String RANK, SUIT;
    private int COLOUR;

    private static final String[] RANKS = {"Ace","2","3","4","5","6","7","8","9","10","Jack","Queen","King"};
    private static final String[] SUITS = {"Clubs","Spades", "Diamonds","Hearts"};
    private static final String[] COLOURS = {"Black","Red"};

    private static final Random GENERATOR = new Random();

    public Card() {
        RANK = RANKS[GENERATOR.nextInt(RANKS.length)];
        SUIT = SUITS[GENERATOR.nextInt(SUITS.length)];
    }
    public Card(String suit, String rank){
        SUIT = suit;
        RANK = rank;
    }
    public String getRANK() {
        return RANK;
    }

    public String getCOLOUR(){
        return COLOURS[COLOUR];
    }

    public String getSUIT() {
        return SUIT;
    }

    public static String[] getAllSuits() {
```

```
35          return SUITS;
36      }
37
38      public static String[] getAllRanks() {
39          return RANKS;
40      }
41
42      public int getRankValue(){
43          int returned = -1;
44          for (int i = 0; i < RANKS.length; i++){
45              if (RANK.equals(RANKS[i])){
46                  returned = i + 1;
47              }
48          }
49          return returned;
50      }
51
52      @Override
53      public String toString(){
54          return getRANK() + " of " + getSUIT();
55      }
56
57      public int compareTo(Card otherCard){
58          if (this.getRankValue() > otherCard.
   getRankValue()) return 1;
59          else if (this.getRankValue() < otherCard.
   getRankValue()) return -1;
60          else return 0;
61      }
62
63  }
64
```

```
 1 public class Menu {
 2     public static void main(String[] args) {
 3         System.out.println("Welcome to Eleven's
   Solitaire!");
 4         System.out.println("Select a menu option to
   begin");
 5         System.out.println("[1] New Game");
 6         System.out.println("[0] Exit");
 7     }
 8
 9
10
11 }
12
```

```java
public class Queue <T> implements QueueInterface<T>{
    private MyNode<T> front;
    private MyNode<T> rear;

    public Queue() {
        front = null;
        rear = null;
    }
    public void enqueue(T newEntry) {
        MyNode<T> newNode = new MyNode<>(newEntry);
        if (front == null){
            front = newNode;
            rear = newNode;
        } else {
            rear.setNext(newNode);
            rear = newNode;
        }

    }


    public T dequeue() {
        if (front == null) return null;
        else {
            T valueToReturn = front.getData();
            front = front.getNext();
            if (front == null) rear = null;
            return valueToReturn;

        }
    }


    public T getFront() {
        if (front == null) return null;
        else return front.getData();
    }


    public boolean isEmpty() {
        return (front == null);
```

```
42      }
43
44
45      public void clear() {
46          front = null;
47          rear = null;
48      }
49
50      public static void main(String[] args) {
51          Queue<String> queue = new Queue<String>();
52
53          queue.enqueue("Milk");
54          queue.enqueue("Eggs");
55          queue.enqueue("Bread");
56
57          for (int i = 0; i <=4; i++){
58              System.out.println("Get Front: " +queue.
    getFront());
59              System.out.println("Dequeue: " + queue.
    dequeue());
60          }
61          queue.enqueue("Cheese");
62          queue.enqueue("Steak");
63          queue.enqueue("Fish");
64
65          System.out.println("Is Queue Empty? : " +
    queue.isEmpty());
66          System.out.println("Clearing Queue");
67          queue.clear();
68          System.out.println("Is Queue Empty? : " +
    queue.isEmpty());
69      }
70  }
71
```

```java
import java.util.EmptyStackException;

public class Stack<T> implements StackInterface<T>{

    private MyNode<T> topNode;

    public Stack(){
        topNode = null;
    }

    public void push(T newEntry) {
        MyNode<T> newNode = new MyNode<>(newEntry);
        newNode.setNext(topNode);
        topNode = newNode;
    }


    public T pop() {
        T dataToReturn = peek();
        topNode = topNode.getNext();
        return dataToReturn;
    }


    public T peek() {
        if (topNode == null) throw new
EmptyStackException();
        else return topNode.getData();
    }


    public boolean isEmpty() {
        return (topNode == null);
    }


    public void clear() {
        topNode = null;
    }

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

```
41          Stack<Integer> stackTest = new Stack<>();
42
43          // Adding Entries
44          stackTest.push(1);
45          stackTest.push(2);
46          stackTest.push(3);
47
48          try {
49              for (int i = 0; i < 4; i++){
50                  System.out.println("Peek: " +
    stackTest.peek());
51                  System.out.println("Pop: " +
    stackTest.pop());
52                  System.out.println("Pop: " +
    stackTest.pop());
53                  System.out.println("Pop: " +
    stackTest.pop());
54                  System.out.println("Pop: " +
    stackTest.pop());
55
56              }
57          }catch (EmptyStackException e){
58              System.out.println("Stack is empty");
59          }
60
61          stackTest.push(4);
62          stackTest.push(5);
63          stackTest.push(6);
64
65          System.out.println("Empty: " + stackTest.
    isEmpty());
66          System.out.println("Clearing stack...");
67          stackTest.clear();
68          System.out.println("Empty: " + stackTest.
    isEmpty());
69
70      }
71  }
72
```

```java
public class MyNode<T> {
    private T data;
    private MyNode<T> next;

    public MyNode(T dataValue) {
        data = dataValue;
        next = null;
    }

    public T getData() {
        return data;
    }

    public void setData(T dataValue){
        data = dataValue;
    }

    public MyNode<T> getNext(){
        return next;
    }

    public void setNext(MyNode<T> nextNode){
        next = nextNode;
    }
    public static void main(String[] args){
        MyNode<Integer> node1 = new MyNode<Integer>(1);
        MyNode<Integer> node2 = new MyNode<Integer>(2);
        MyNode<Integer> node3 = new MyNode<Integer>(3);

        node1.setNext(node2);
        node2.setNext(node3);

        System.out.println("Node 1 is: " + node1.getData());
        System.out.println("Node 2 is: " + node1.getNext().getData());
        System.out.println("Node 3 is: " + node1.getNext().getNext().getData());
```

```
36
37      }
38
39 }
40
```

```java
import java.sql.SQLOutput;
import java.util.EmptyStackException;
import java.util.InputMismatchException;
import java.util.Scanner;

public class Elevens {
    // Creating a shuffled stack of cards to be used in the game
    private static Stack<Card> playDeck;

    private static final AListArray<Card> playedDeck = new AListArray<Card>();
    private static CardDeck deck = new CardDeck();
    // When a card is removed from the pulled cards, it is added to this stack
    private static final Stack<Card> usedCards = new Stack<Card>();

    // Queue of recorded moves & a copy of the original deck used so the computer can replay the users game later
    private static final Queue<Integer> recordedMoves = new Queue<Integer>();
    private static final Queue<Card> recordedCards = new Queue<Card>();
    private static Stack<Card> savedDeck = new Stack<Card>();

    private static int moveCounter = 0;
    private static int hintsUsed = 0;
    private static final int TARGET = 11;

    private static void startGame(){
        // Initialising Deck of cards
        deck = new CardDeck();
        playDeck = deck.shuffle(deck.toArray());
        savedDeck = playDeck;

        boolean finished = false;
        boolean stalemate = false;
        boolean victory = false;
```

```
33          moveCounter = 0;
34          Card card1;
35          Card card2;
36          Card card3;
37          dealCards(playDeck);
38          while (!finished && !playedDeck.isEmpty()){
39              stalemate = isStalemate();
40              victory = isVictory();
41              if (victory){
42                  System.out.println("You cleared the
    deck. Congratulations");
43                  System.out.println("Moves Made: " +
    moveCounter);
44                  System.out.println("Hints Used: " +
    hintsUsed);
45              }
46              if (stalemate){
47                  printDeck();
48                  System.out.println("You have reached
    a stalemate | You Lose");
49                  System.out.println("Moves Made: " +
    moveCounter);
50                  System.out.println("Hints Used: " +
    hintsUsed);
51                  System.out.println("Press [1] to
    start a new game | [2] Replay |[0] Quit");
52                  int option = selectCard();
53                  if (option == 1) {
54                      finished = false;
55                      hintsUsed = 0;
56                      recordedMoves.clear();
57                      playedDeck.clear();
58                      startGame();
59                  } else if (option == 2){
60                      replayGame();
61                  }else {
62                      finished = true;
63                      break;
64                  }
65              }
66              // If the stack is empty the game is won
```

```
66 and a message is printed
67             if (playedDeck.isEmpty()) System.out.
   println("You Cleared the Deck! Congratulations ");
68             printDeck();
69             System.out.println("[Moves made: " +
   moveCounter + " Hints Used: " + hintsUsed + "]");
70
71             System.out.println("Select your first
   card or press 0 for a Hint");
72             int option = selectCard();
73             if (option == 0) giveHint(findMove());
74             else {
75                 // Selecting the first card
76                 int c1Pos = option;
77                 card1 = playedDeck.getEntry(c1Pos);
78
79                 // Selecting the second card
80                 int c2Pos = selectCard();
81                 card2 = playedDeck.getEntry(c2Pos);
82                 while (c2Pos == c1Pos){
83                     System.out.println("You cannot
   select two of the same cards | Try Again");
84                     c2Pos = selectCard();
85                     card2 = playedDeck.getEntry(
   c2Pos);
86
87                 }
88
89                 // If the values of the 2 cards = 11
   , remove & replace them
90                 if (card1.getRankValue() + card2.
   getRankValue() == TARGET){
91                     recordedMoves.enqueue(c1Pos);
92                     recordedMoves.enqueue(c2Pos);
93                     recordedCards.enqueue(card1);
94                     recordedCards.enqueue(card2);
95
96                     System.out.println("2 Cards
   removed");
97                     removeCard(c1Pos, playDeck);
98                     removeCard(c2Pos, playDeck);
```

```
 99
100                     moveCounter++;
101
102                 } else if (card1.getRankValue() +
    card2.getRankValue() > 21){
103                     int c3Pos = selectCard();
104                     card3 = playedDeck.getEntry(
    c3Pos);
105                     if (card1.getRankValue() + card2
    .getRankValue() + card3.getRankValue() == 36){
106                         recordedMoves.enqueue(c1Pos
    );
107                         recordedMoves.enqueue(c2Pos
    );
108                         recordedMoves.enqueue(c3Pos
    );
109
110                         recordedCards.enqueue(card1
    );
111                         recordedCards.enqueue(card2
    );
112                         recordedCards.enqueue(card3
    );
113
114                         System.out.println("3 Cards
    removed");
115                         removeCard(c1Pos, playDeck);
116                         removeCard(c2Pos, playDeck);
117                         removeCard(c3Pos, playDeck);
118
119
120                         moveCounter++;
121                     }
122                 }
123                 else System.out.println("Cards do
    not equal 11 | Try Again");
124             }
125         }
126
127
128
```

```
129
130         /*while (!finished){
131             System.out.println("Welcome to Eleven's
    Solitaire!");
132             System.out.print("Moves made: " +
    moveCounter);
133         } */
134     }
135
136     private static void demoMode(){
137         // Initialising Deck of cards
138         deck = new CardDeck();
139         playDeck = deck.shuffle(deck.toArray());
140
141         boolean finished = false;
142         boolean stalemate = false;
143         boolean victory = false;
144         int[] possibleMove;
145         moveCounter = 0;
146         Card card1;
147         Card card2;
148         Card card3;
149         dealCards(playDeck);
150
151         while (!finished && !playedDeck.isEmpty()) {
152             printDeck();
153             stalemate = isStalemate();
154             victory = isVictory();
155
156             if (victory){
157                 System.out.println("The computer
    beat the game!");
158                 System.out.println("[1] Back to menu
    ");
159                 System.out.println("[0] Quit Game");
160                 int input = getInput();
161                 if (input == 1) menu();
162                 else {
163                     finished = true;
164                     break;
165                 }
```

```
166             } if (stalemate){
167                 System.out.println("A stalemate was
    reached");
168                 System.out.println("[1] Back to menu
    ");
169                 System.out.println("[0] Quit Game");
170                 int input = getInput();
171                 if (input == 1) menu();
172                 else {
173                     finished = true;
174                     break;
175                 }
176             }
177             if (!isThreeFaces()){
178                 card1 = findMove()[0];
179                 card2 = findMove()[1];
180                 removeCard(findMove()[0], playDeck);
181                 removeCard(findMove()[1], playDeck);
182             } else {
183                 card1 = findMove()[0];
184                 card2 = findMove()[1];
185                 card3 = findMove()[2];
186                 removeCard(findMove()[0], playDeck);
187                 removeCard(findMove()[1], playDeck);
188                 removeCard(findMove()[2], playDeck);
189             }
190         }
191     }
192
193     private static void removeCard(int pos, Stack<
    Card> deck){
194         try {
195             playedDeck.replace(pos,deck.pop());
196         }catch (EmptyStackException e) {
197             playedDeck.remove(pos);
198         }
199     }
200     private static void removeCard(Card card, Stack<
    Card> deck){
201         int pos  = playedDeck.getPos(card);
202         try {
```

```
203             playedDeck.replace(pos,deck.pop());
204         }catch (EmptyStackException e) {
205             playedDeck.remove(pos);
206         }
207     }
208 
209     private static void replayGame(){
210         int input = -1;
211         dealCards(savedDeck);
212         Card card1;
213         Card card2;
214         Card card3;
215 
216         while (input != 0){
217             printDeck();
218             try {
219                 System.out.println("Press [1] to
    goto next move: ");
220                 System.out.println("Press [0] quit
    to menu");
221                 input = getInput();
222                 card1 = recordedCards.dequeue();
223                 if (card1.getRankValue() > 10){
224                     card2 = recordedCards.dequeue();
225                     card3 = recordedCards.dequeue();
226 
227                     removeCard(recordedMoves.dequeue
    (), savedDeck);
228                     removeCard(recordedMoves.dequeue
    (), savedDeck);
229                     removeCard(recordedMoves.dequeue
    (), savedDeck);
230 
231                     System.out.println(card1 + "
    Removed");
232                     System.out.println(card2 + "
    Removed");
233                     System.out.println(card3 + "
    Removed");
234                 }else {
235                     card2 = recordedCards.dequeue();
```

```
236                     removeCard(recordedMoves.dequeue
    (), savedDeck);
237                     removeCard(recordedMoves.dequeue
    (), savedDeck);
238                     System.out.println(card1 + "
    Removed");
239                     System.out.println(card2 + "
    Removed");
240                 }if (input == 0){
241                     menu();
242                 }
243             }catch (NullPointerException e){
244                 input = 0;
245                 System.out.println("That was your
    last move");
246                 System.out.println("[1] Goto Menu [2
    ] Quit");
247                 int option = getInput();
248                 if (option != 2) menu();
249                 else break;
250             }
251         }
252     }
253
254     private static boolean isVictory(){
255         return playedDeck.isEmpty();
256     }
257
258     private static void dealCards(Stack<Card>
    cardStack){
259         for (int  i = 0; i < 9; i++){
260             playedDeck.add(cardStack.pop());
261         }
262     }
263     private static void printDeck(){
264         for (int i = 1; i <= 3; i++){
265             try {
266                 System.out.print("[" + (i) + "]" +
    " " + playedDeck.getEntry(i) + " ");
267             }catch (IndexOutOfBoundsException e){
268                 System.out.println("[EMPTY]");
```

```
269             }
270         }
271         System.out.println("");
272         for (int i = 4; i <= 6; i++){
273             try {
274                 System.out.print("[" + (i) + "]" +
    " " + playedDeck.getEntry(i) + " ");
275             }catch (IndexOutOfBoundsException e){
276                 System.out.println("[EMPTY]");
277             }
278         }
279         System.out.println("");
280         for (int i = 7; i <= 9; i++){
281             try {
282                 System.out.print("[" + (i) + "]" +
    " " + playedDeck.getEntry(i) + " ");
283             }catch (IndexOutOfBoundsException e){
284                 System.out.println("[EMPTY]");
285             }
286         }
287         System.out.println("");
288     }
289
290     private static int selectCard(){
291         System.out.println("Select: ");
292         int input = getInput();
293         return input;
294     }
295
296     // This method checks if it is possible to
    remove three faces at once
297     private static boolean isThreeFaces(){
298         boolean kings = false;
299         boolean jacks = false;
300         boolean queens = false;
301
302         for (int i = 1; i < playedDeck.getLength();
    i++) {
303             switch (playedDeck.getEntry(i).
    getRankValue()){
304                 case 11:
```

```
305                     jacks=true;
306                     break;
307                 case 12:
308                     queens=true;
309                     break;
310                 case 13:
311                     kings=true;
312                     break;
313             }
314 
315         }
316         return kings && queens && jacks;
317     }
318 
319     private static Card[] findMove(){
320         Card[] clearTwo = new Card[2];
321         Card[] clearThree = new Card[3];
322         boolean is11 = false;
323         Card card1 = new Card();
324         Card card2 = new Card();
325         Card card3 = new Card();
326 
327         // if it is possible to remove three cards
    at once then it makes sense to make this move first
328         if (isThreeFaces()){
329             for (int i = 1; i < playedDeck.getLength
    (); i++) {
330                 switch (playedDeck.getEntry(i).
    getRankValue()){
331                     case 11:
332                         card1 = playedDeck.getEntry(
    i);
333                     case 12:
334                         card2 = playedDeck.getEntry(
    i);
335                     case 13:
336                         card3 = playedDeck.getEntry(
    i);
337                 }
338             }
339             clearThree[0] = card1;
```

```
340             clearThree[1] = card2;
341             clearThree[2] = card3;
342             return clearThree;
343         } else {
344             for (int i = 1; i < playedDeck.getLength
    (); i++) {
345                 for (int j = 1; j < playedDeck.
    getLength(); j++) {
346                     // If statement to ensure the
    method does not add two duplicate cards together
347                     if (playedDeck.getEntry(i).
    getRankValue() != playedDeck.getEntry(j).
    getRankValue()) {
348                         // if two values added = 11
    then there is no stalemate as a move is possible
349                         is11 = playedDeck.getEntry(i
    ).getRankValue() + playedDeck.getEntry(j).
    getRankValue() == 11;
350                         if (is11){
351                             System.out.println(
    playedDeck.getEntry(i));
352                             System.out.println(
    playedDeck.getEntry(j));
353                             clearTwo[0] = playedDeck
    .getEntry(i);
354                             clearTwo[1] = playedDeck
    .getEntry(j);
355                             return clearTwo;
356                         }
357                     }
358                 }
359             }
360         }
361         return clearTwo;
362     }
363
364     private static void giveHint(Card[] moveArr){
365         hintsUsed++;
366         String hint;
367         if (isThreeFaces()) System.out.println("
    Reminder! If a King, Queen & Jack are present, all
```

```
367 three may be removed");
368         else {
369             moveArr = findMove();
370             System.out.println(moveArr[0] + " " +
    moveArr[1]);
371             hint =  moveArr[0] + " + x = 11?";
372             System.out.println(hint);
373         }
374     }
375 
376     // This method checks if there is a stalemate (
    No moves are possible)
377     private static boolean isStalemate(){
378         // Initialising is11 variable
379         boolean is11 = false;
380 
381             // For loop to iterate through the array
     adding each pair of values together
382             for (int i = 1; i < playedDeck.getLength
    (); i++) {
383                 for (int j = 1; j < playedDeck.
    getLength(); j++) {
384                     // If statement to ensure the
    method does not add two duplicate cards together
385                     if (playedDeck.getEntry(i).
    getRankValue() != playedDeck.getEntry(j).
    getRankValue()) {
386                         // if two values added = 11
    then there is no stalemate as a move is possible
387                         is11 = playedDeck.getEntry(i
    ).getRankValue() + playedDeck.getEntry(j).
    getRankValue() == 11;
388                         if (is11){
389                             return false;
390                         }else {
391                             //If there is no pair
    that add to 11, check if it is possible to remove 3
    faces
392                             if (isThreeFaces()){
393                                 return false;
394                             }
```

```
395                             }
396                         }
397                     }
398                 }
399                 // if the is11 variable is not updated
    to true  or there are no 3 faces
400                 // then no moves are possible and a
    stalemate is reached
401             return true;
402         }
403
404         private static void menu(){
405             System.out.println("Welcome to Eleven's
    Solitaire!");
406             System.out.println("Select a menu option to
    begin");
407             System.out.println("[1] New Game");
408             System.out.println("[2] Demonstration Mode"
    );
409             System.out.println("[0] Exit");
410             int input = getInput();
411             switch (input) {
412                 case 1 -> startGame();
413                 case 2 -> demoMode();
414                 case 0 -> System.exit(0);
415             }
416
417         }
418
419         private static int getInput(){
420             Scanner scan = new Scanner(System.in);
421             boolean valid = false;
422             int option = -1;
423             while (!valid){
424                 try{
425                     option = scan.nextInt();
426                     valid = true;
427                 }catch (InputMismatchException e){
428                     System.out.println("Invalid Input");
429                     option = scan.nextInt();
430                 }
```

```
431             }
432             return option;
433         }
434
435         public static void main(String[] args) {
436             Elevens el = new Elevens();
437             menu();
438         }
439
440
441 }
442
```

```java
import java.util.Random;

public class CardDeck {
    private static final Card[] cardDeck = new Card[52];
    private static  Stack<Card> DeckOfCards;
    private static final String[] suitList = Card.getAllSuits();
    private static final String[] rankList = Card.getAllRanks();
    private static final Random RANDOM = new Random();

    public CardDeck() {
        DeckOfCards = new Stack<Card>();
        for (int i = 0; i < suitList.length; i++){
            for (int j = 0; j < rankList.length; j++){
                DeckOfCards.push(new Card(suitList[i], rankList[j]));
            }
        }


    }

    // This method takes an array of cards, shuffles them and returns them as a stack
    // ready to be used in the game
    public Stack<Card> shuffle(Card[] array){
        AListArray<Integer> tmpArray = new AListArray<Integer>();
        int genNum = 0;
        int counter = 0;
        boolean isUnique = false;
        Stack<Card> shuffled = new Stack<Card>();
        int[] nTmpA = new int[52];
        int index, temp;
        for (int i = 0; i < nTmpA.length; i++){
            nTmpA[i] = i;
        }
```

```
34
35          for (int i = nTmpA.length - 1; i > 0; i--)
36          {
37              index = RANDOM.nextInt(i + 1);
38              temp = nTmpA[index];
39              nTmpA[index] = nTmpA[i];
40              nTmpA[i] = temp;
41          }
42
43          for (int i  = 0; i < array.length; i++){
44              shuffled.push(array[nTmpA[i]]);
45          }
46          return shuffled;
47      }
48
49      public Card[] toArray(){
50          Card[] arr = new Card[52];
51          for (int i = 0; i < arr.length; i++){
52              arr[i] = DeckOfCards.pop();
53          }
54          return arr;
55      }
56 }
57
```

```
 1 public class CardTest {
 2     private static final Card[] cardDeck = new Card[
   52];
 3     private static final String[] suitList = Card.
   getAllSuits();
 4     private static final String[] rankList = Card.
   getAllRanks();
 5
 6     // Creating a shuffled stack of cards to be used
   in the game
 7     private static Stack<Card> playDeck;
 8
 9     private static final AListArray<Card> playedDeck
    = new AListArray<Card>();
10     private static CardDeck deck = new CardDeck();
11     // When a card is removed from the pulled cards,
   it is added to this stack
12     private static final Stack<Card> usedCards = new
   Stack<Card>();
13
14     // Queue of recorded moves & a copy of the
   original deck used so the computer can replay the
   users game later
15     private static final Queue<Integer> recordedMoves
    = new Queue<Integer>();
16     private static final Queue<Card> recordedCards =
   new Queue<Card>();
17     private static Stack<Card> savedDeck = new Stack<
   Card>();
18
19     private static int moveCounter = 0;
20     private static int hintsUsed = 0;
21     private static final int TARGET = 11;
22
23     /*
24         private static Card[] findMove(){
25         int[] clearTwo = new int[2];
26         int[] clearThree = new int[3];
27         boolean is11 = false;
28         int card1 = -1;
29         int card2 = -1;
```

```
30          int card3 = -1;
31
32          // if it is possible to remove three cards at
     once then it makes sense to make this move first
33          if (isThreeFaces()){
34              for (int i = 1; i < playedDeck.getLength
    (); i++) {
35                  switch (playedDeck.getEntry(i).
    getRankValue()){
36                      case 11:
37                          card1 = playedDeck.getEntry(i
    ).getRankValue();
38                      case 12:
39                          card2 = playedDeck.getEntry(i
    ).getRankValue();
40                      case 13:
41                          card3 = playedDeck.getEntry(i
    ).getRankValue();
42                  }
43              }
44              clearThree[0] = card1;
45              clearThree[1] = card2;
46              clearThree[2] = card3;
47              return clearThree;
48          } else {
49              for (int i = 1; i < playedDeck.getLength
    (); i++) {
50                  for (int j = 1; j < playedDeck.
    getLength(); j++) {
51                      // If statement to ensure the
    method does not add two duplicate cards together
52                      if (playedDeck.getEntry(i).
    getRankValue() != playedDeck.getEntry(j).getRankValue
    ()) {
53                          // if two values added = 11
    then there is no stalemate as a move is possible
54                          is11 = playedDeck.getEntry(i
    ).getRankValue() + playedDeck.getEntry(j).
    getRankValue() == 11;
55                          if (is11){
56                              System.out.println(
```

```
56 playedDeck.getEntry(i));
57                                 System.out.println(
   playedDeck.getEntry(j));
58                                 clearTwo[0] = playedDeck.
   getEntry(i).getRankValue();
59                                 clearTwo[1] = playedDeck.
   getEntry(j).getRankValue();
60                                 return clearTwo;
61                             }
62                         }
63                     }
64                 }
65             }
66             return clearTwo;
67         }
68          */
69         public static void main(String[] args) {
70             Card card1 = new Card();
71             String[] list = card1.getAllSuits();
72             System.out.println(list[0]);
73 
74             CardDeck deck = new CardDeck();
75             Card[] cardDeck = deck.toArray();
76 
77 
78             Stack<Card> shuffledDeck = deck.shuffle(
   cardDeck);
79 
80             for (int i = 0; i < cardDeck.length; i++){
81                 System.out.println(shuffledDeck.pop());
82             }
83 
84 
85         }
86 }
87 
```

```java
import java.util.Arrays;

public class AListArray <T> implements ListInterface <T>{

    private T[] list;
    private int numEntries;
    private int capacity;
    private final int DEFAULT_SIZE = 100;

    private void addCapacity(){
        capacity += DEFAULT_SIZE;
        list = Arrays.copyOf(list, capacity +1);
    }

    public AListArray(){
        T[] tempList =(T[]) new Object[DEFAULT_SIZE + 1];
        numEntries = 0;
        list = tempList;
        capacity = DEFAULT_SIZE;
    }
    @Override
    public void add(T newEntry) {
        if (numEntries == capacity) addCapacity();
        numEntries++;
        list[numEntries] = newEntry;
    }

    @Override
    public void add(int newPos, T newEntry) {
        if (newPos >= 1 && newPos <= numEntries +1){
            if (numEntries == capacity) addCapacity();
            for (int i = numEntries; i >= newPos; i--) list[i+1] = list[i];
            list[newPos] = newEntry;
            numEntries++;
        } else throw new IndexOutOfBoundsException("Error: OUT OF BOUNDS");

```

```
37      }
38
39      @Override
40      public T remove(int position) {
41          if (position >= 1 && position <= numEntries){
42              T valueToReturn = list[position];
43              for (int i = position; i < numEntries; i
    ++) list[i] = list[i + 1];
44              numEntries--;
45              return valueToReturn;
46          } else throw new IndexOutOfBoundsException("
    Error: OUT OF BOUNDS");
47      }
48
49      @Override
50      public void clear() {
51          numEntries = 0;
52      }
53
54      @Override
55      public T replace(int position, T newEntry) {
56          if (position >= 1 && position <= numEntries){
57              T valueToReturn = list[position];
58              list[position] = newEntry;
59              return valueToReturn;
60          } else throw new IndexOutOfBoundsException("
    Error: OUT OF BOUNDS");
61      }
62
63      @Override
64      public T getEntry(int position) {
65          if (position >= 1 && position <= numEntries){
66              return list[position];
67          } else throw new IndexOutOfBoundsException("
    Error: OUT OF BOUNDS");
68      }
69
70      public T[] toArray(){
71          T[] array = (T[]) new Object[numEntries];
72          System.arraycopy(list, 1, array, 0,
    numEntries);
```

```
 73          return array;
 74      }
 75
 76      @Override
 77      public boolean contains(T anEntry) {
 78          boolean found = false;
 79          int i = 1;
 80          while (i++ <= numEntries && !found)
 81              if (list[i].equals(anEntry)) found =true
    ;
 82          return found;
 83      }
 84
 85      // Method to return the position of an entry in
    the array
 86      public int getPos(T anEntry) {
 87          // Setting the index to an impossible value
    to indicate an error if it is returned in this form
 88          int index = -1;
 89          int i = 1;
 90          while (i++ <= numEntries)
 91              if (list[i].equals(anEntry)) index = i;
 92          System.out.println(index);
 93          return index;
 94      }
 95
 96      public int getLength(){
 97          return numEntries;
 98      }
 99
100      @Override
101      public boolean isEmpty() {
102          return numEntries == 0;
103      }
104  }
105
```

```java
public interface ListInterface<T> {

    public void add(T newEntry);


    public void add(int newPos, T newEntry);


    public T remove(int position);

    public void clear();


    public T replace(int position, T newEntry);

    public T getEntry(int position);

    public T[] toArray();

    public boolean contains(T anEntry);

    public int getLength();


    public boolean isEmpty();
}

```

```java
public interface QueueInterface <T> {

    public void enqueue(T newEntry);
    public T dequeue();
    public T getFront();
    public boolean isEmpty();
    public void clear();

}

```

```java
public interface StackInterface<T>{
    public void push(T newEntry);

    public T pop();

    public T peek();

    public boolean isEmpty();

    public void clear();
}

```