



Java Source Code - Elevens

Author

Mr. Adam Torok - B00798824

JORDANSTOWN, APRIL 2021

App.java

```
1 package arch;
2
3 import java.util.Scanner;
4
5 public class App {
6     Scanner scan = new Scanner(System.in);
7     private int gameState = 0; //0 running game | 1 win | -1 staleMate
8     private Replay replay = new Replay();
9
10
11     public App() throws LockedDeckException, EmptyDeckException {}
12
13     public void selectMode() throws LockedDeckException, EmptyDeckException { //select and run the
14         mode
15         int user;
16
17         System.out.println("CardGame\n-----");
18         System.out.println("1. Play\n2. Demonstration Mode\n3. Rules\n4. Exit");
19         Scanner scan = new Scanner(System.in); //this is for testing
20         user = Integer.parseInt(scan.nextLine());
21
22         //until we get a valid choice
23         //call the corresponding mode
24         while (user != 4) {
25             if (user == 1) {
26                 System.out.println("Play Mode\n");
27                 playMode(false);
28             } else if (user == 2) {
29                 System.out.println("Demonstration Mode");
30                 playMode(true);
31             } else if (user == 3) {
32                 System.out.println("The Rules of the Game\n");
33                 showRules();
34             } else if (user == 4) {
35                 System.out.println("Bye...");
36             }
37         }
38     }
39
40     //main game
41     public void playMode(boolean boo) throws LockedDeckException, EmptyDeckException {
42         System.out.println("Dealing Your cards.....");
43         //get a new deck and board
44         Deck d = new Deck(true, true);
45         Board b = new Board();
46         //deal the BOARDSIZE amount of cards
47         for (int i = 0; i < b.BOARDSIZE; i++) {
48             b.addNewEntry(d.removeFirstElement());
49         }
50         //print out the board
51         System.out.println(b.representBoard());
52         System.out.println("-----");
53         String initBoard = "Initial board: " + b.toString();
54         ReplayItem r = new ReplayItem(initBoard);
55         replay.push(r);
56
57         //until no valid moves or (deck and board
58         while (gameState != -1 || (d.getSize() == 0 && b.getSize() == 0)) {
59             //gets empty
60             if (gameState == -1) {
61                 break; //break out
62             }
63             //clear the valid moves stack
64             b.getValidMove().clear();
65             int cardRemoved = 0;
66             boolean acceptableMove = false;
67
68             //get all valid moves
```

```
68     Stack v = b.getValidMove();
69     int[] choice = {0, 0, 0};
70     //if the valid stack has moves
71     if (v.getSize() > 0) {
72         if (boo == true) {
73             //the machine will auto peek to learn a posish
74             choice = v.peek();
75         } else { //USER MODE
76             //user interface, hint, validate
77             boolean proceed = false;
78             while (proceed == false) {
79                 System.out.print("[ " + d.getSize() + "]]]]\tPlease Select Card: ");
80                 String user = scan.nextLine();
81                 if (user.length() > 0 && user.length() <= 3 && !user.equals("h")) {
82                     for (int i = 0; i <= user.length() - 1; i++) {
83                         choice[i] = user.charAt(i) - 48;
84                     }
85                     //order the users input
86                     int[] tmpArr = choice;
87                     for (int i = 0; i <= tmpArr.length; i++) {
88                         for (int j = i + 1; j < tmpArr.length; j++) {
89                             int tmp = 0;
90                             if (tmpArr[i] < tmpArr[j]) {
91                                 tmp = tmpArr[i];
92                                 tmpArr[i] = tmpArr[j];
93                                 tmpArr[j] = tmp;
94                             }
95                         }
96                     }
97                     //validate their selection
98                     if (!b.checkAnswer(choice[0], choice[1]) &&
99                         b.checkAnswer(choice[0], choice[1], choice[2]) == false) {
100                         System.out.println("This is not good selection");
101                     } else {
102                         //invalid input will stop and ask again
103                         proceed = true;
104                         break;
105                     }
106                     //activate hint & auto remove
107                 } else if (user.equals("h")) {
108                     choice = v.peek();
109                     String hint = "";
110                     for (int i = 0; i < choice.length; i++) {
111                         hint += Integer.toString(choice[i]) + " ";
112                     }
113                     System.out.println("You can continue like this -> " + hint);
114                     System.out.println("Let me help ya, I remove it for you...");
115                     proceed = true;
116                     break;
117                 }
118             }
119         }
120         //extra security, validating the move on
121         //the board as well | input nn0
122         if (choice.length == 2 || choice[2] == 0) {
123             cardRemoved = 2;
124             acceptableMove = b.checkAnswer(choice[0], choice[1]);
125             //testing input nnn
126         } else if (choice.length == 3) {
127             cardRemoved = 3;
128             acceptableMove = b.checkAnswer(choice[0], choice[1], choice[2]);
129         } else {
130             gameState = -1;
131             break;
132         }
133         //no valid moves in the stack so stalemate
134     } else {
135         gameState = -1;
136         System.out.println("No valid Moves");
137         break;
138     }
139 }
```

```
138     }
139     //if the users move is acceptable
140     if (acceptableMove) {
141         //prepare the input for replay
142         String choosen = "";
143         choosen += b.getNthCard(choice[0]).toString() + " ";
144         choosen += b.getNthCard(choice[1]).toString() + " ";
145         if (choice.length == 3 && choice[2] != 0) {
146             choosen += b.getNthCard(choice[2]).toString() + " ";
147         }
148         //reprint the selection
149         System.out.println("Removed Cards:\t" + choosen);
150
151         //solve the KJQ tab issue
152         if (b.getSize() > 0) {
153             if (choice.length == 3) {
154                 choosen += "\tafter-->\t" + b.toString();
155             } else {
156                 choosen += "\t\tafter-->\t" + b.toString();
157             }
158         }
159
160         //push the selection to the stack
161         ReplayItem rinit = new ReplayItem(choosen);
162         replay.push(rinit);
163
164         if (b.getSize() == 2) {
165             //artificial board clear when two cards left
166             //anytime it happens, that is a won scen.
167             // to avoid mess up with the replay
168             b.clear();
169         } else {
170             for (int i : choice) {
171                 b.removeNthCard(i);
172             }
173         }
174         //if deck has 2 card we cant remove 3
175         if (d.getSize() > 0) {
176             int toReplace = Math.min(d.getSize(), cardRemoved);
177             //replace the cards
178             for (int i = 0; i < toReplace; i++) {
179                 b.addNewEntry(d.removeFirstElement());
180             }
181         }
182         //no deck, no board means a win
183         if (d.getSize() == 0 && b.getSize() == 0) {
184             gameState = 1;
185             break;
186         }
187         //display the empty board
188         System.out.println("-----");
189         System.out.println(b.representBoard());
190     }
191 }
192 //things relating to the game state
193 switch (gameState) {
194     //the game is over
195     case -1:
196         System.out.println("\n\"Game over man, GAME OVER!\" - Pvt. Hudson");
197         System.out.println("Cards Left: " + d.getSize());
198         break;
199     //win
200     //create a new board to proper cleanup
201     case 1:
202         System.out.println("\nHold my beer..... YOU WON\n\n");
203         b = new Board();
204 }
205 //handling replay and quit
206 System.out.println("\nWould you like to see the replay? (y/n) - (q) quit");
207 String choice = scan.next();
```

```
208 //render replay
209 if (choice.equals("y") || choice.equals("Y")) {
210     String finalBoard = "Final Board:\t" + b.toString();
211
212     System.out.println("Action Replay!");
213     System.out.println("Replay Size: " + replay.size());
214     replay.push(new ReplayItem(finalBoard));
215     replay.printHistory();
216 //quit
217 } else if (choice.equals("q") || choice.equals("Q")
218           || choice.equals("n") || choice.equals("N")) {
219     System.exit(0);
220 }
221 }
222 //nice touch!
223 public static void showRules() {
224     System.out.println("Elevens is extremely similar to Bowling Solitaire,\n" +
225                       "except that the layout is a little different and\n" +
226                       "the goal is to make matching pairs that add up to\n" +
227                       "11 rather than adding matching pairs up to 10.\n" +
228                       "\n" +
229                       "Empty spaces in the 9-card formation are automatically\n" +
230                       "filled by placing a card from the Deck in the free space.\n" +
231                       "Once you run out of cards in the Deck, do not fill the\n" +
232                       "empty spaces in the card formation with any other cards.\n" +
233                       "\n" +
234                       "To play this game, look at your 9-card formation and see\n" +
235                       "if any cards can be matched that add up to 11 in total.\n" +
236                       "If you have a matching pair that can create this sum, \n" +
237                       "then you may remove them from place. Once youve done so,\n" +
238                       "remember to fill in the gaps left by these two cards with\n" +
239                       "two cards from the Deck.\n" +
240                       "\n" +
241                       "Only cards in the 9-card formation are available to play\n" +
242                       "with, and you may not build any cards on top of each other\n" +
243                       "during the game. Cards cannot be removed from the Deck \n" +
244                       "unless they are being placed in the table layout, and you \n" +
245                       "should not look at the cards in the Deck before moving them\n" +
246                       "into play. They must remain unknown until they are flipped\n" +
247                       "over to be placed in the 9-card formation.\n" +
248                       "\n" +
249                       "The ranking of cards matches their face value i.e. the two of\n" +
250                       "clubs is equal to two. Aces hold a value of one and Jacks, \n" +
251                       "Queens, and Kings equal eleven only when they are removed \n" +
252                       "together. For example, if you have a Jack and King on your \n" +
253                       "board you cant remove either until a Queen appears. Once all\n" +
254                       "three cards are present on the board they can be removed \n" +
255                       "together to make 11. They are the only cards in the game \n" +
256                       "that are moved as a trio, rather than being matched as a pair.\n" +
257                       "\n" +
258                       "HOW TO WIN:\n" +
259                       "\n" +
260                       "To win at a round of Elevens, you must remove absolutely all\n" +
261                       "cards from play including those from the Deck. Once you \n" +
262                       "have matched all cards in the Deck, then you have won the round.\n" +
263                       "\n" +
264                       "It is possible to play this game with more than one player. \n" +
265                       "To do so, you could create a scoring system by having each \n" +
266                       "player keep their matched pairs and making each set worth 1 point.\n" +
267                       "The player with the highest number of points would win the game.\n" +
268                       "Typically, this is a solo player game, but its extremely easy to\n" +
269                       "make into a family-friendly or party game.");
270 }
271
272 public static void main(String[] args) throws LockedDeckException, EmptyDeckException {
273     App a = new App();
274     try {
275         a.selectMode();
276     } catch (NumberFormatException e) {
277         System.out.println("Invalid Output! Lets try that again");
278     }
279 }
```

```
278         a.selectMode();
279     } catch (LockedDeckException e) {
280         e.printStackTrace();
281     } catch (EmptyDeckException e) {
282         e.printStackTrace();
283     }
284 }
285 }
```

Board.java

```
1 package arch;
2
3 public class Board extends LinkedList<Card> implements CardInterFace {
4     private Card firstCard, lastCard;
5     private Stack validMove = new Stack();
6     int cJ = 0, cQ = 0, cK = 0;
7     public final int BOARDSIZE = 9;
8
9     //just create an empty deck
10    public Board(boolean b) throws LockedDeckException, NullPointerException {
11        firstCard = null;
12        lastCard = null;
13    }
14
15    public Board() {
16        firstCard = null;
17        lastCard = null;
18    }
19
20    //add a new card to the board
21    public boolean addNewEntry(Card newEntry) {
22        Card newCard = newEntry;
23        newCard.setNext(firstCard);
24        firstCard = newCard;
25        size++;
26
27        //if we have only one node, we record it as a lastCard
28        if (getSize() == 1) {
29            lastCard = newCard;
30            lastCard.setNext(null);
31        }
32        //update KJQ counter
33        switch (newEntry.getCardValue()) {
34            case 11:
35                cJ++;
36                break;
37            case 12:
38                cQ++;
39                break;
40            case 13:
41                cK++;
42                break;
43        }
44        sort();
45        return true;
46    }
47
48    public void sort() {
49        if (size > 2) {
50            //bubble sort the board
51            for (int i = 0; i < size - 1; i++) {
52                Card currentCard = firstCard;
53                Card nextCard = (Card) firstCard.getNext();
54                Card prevCard = null;
55                for (int j = 1; j < size; j++) {
56                    if (j == 1) {
57                        prevCard = firstCard;
58                    }
59                    //if the current card value is greater
60                    if (currentCard.getCardValue() > nextCard.getCardValue()) {
61                        String tmpSuit = currentCard.getStrSuit();
62                        int tmpValue = currentCard.getCardValue();
63                        //swap the cards
64                        currentCard.setCardValue(nextCard.getCardValue());
65                        currentCard.setCardSuit(nextCard.getStrSuit());
66                        nextCard.setCardValue(tmpValue);
67                        nextCard.setCardSuit(tmpSuit);
68                        //currentCard.setNext(nextCard.getNext());
69                    }
70                }
71            }
72        }
73    }
74 }
```

```
69         }
70
71         //slipping the prevCard
72         //slipping the current and nexCards
73         prevCard = currentCard;
74         currentCard = nextCard;
75         nextCard = (Card) nextCard.getNext();
76
77     }
78 }
79 }
80 }
81
82 //get an exact cards position
83 public int getCardPosition(Card c) {
84     int i = 1;
85     Card currentCard = firstCard;
86     while (currentCard.getNext() != null) {
87
88         if (currentCard.getCardValue() == c.getCardValue() &&
89             currentCard.getStrSuit() == c.getStrSuit()) {
90             return i;
91         }
92
93         if (currentCard.getCardValue() == c.getCardValue() &&
94             "*" == c.getStrSuit()) {
95             return i;
96         }
97
98         currentCard = (Card) currentCard.getNext();
99         i++;
100     }
101     if (lastCard.getCardValue() == c.getCardValue() &&
102         lastCard.getStrSuit() == c.getStrSuit()) {
103         return i++;
104     }
105     return -1;
106 }
107 //array representation of the board
108 public Card[] toArray() throws NullPointerException {
109     Card[] cardArray = new Card[size];
110     Card[] emptyArray = new Card[0];
111
112     if (isEmpty() == true)
113         throw new NullPointerException("Board is empty");
114
115     int counter = 0;
116     Card currentCard = firstCard;
117     while (currentCard.getNext() != null) {
118         cardArray[counter] = currentCard;
119         currentCard = (Card) currentCard.getNext();
120         counter++;
121     }
122     //cardArray[size-1] = (Card) getLastCard();
123     return cardArray;
124 }
125
126 //string representation of the array
127 public String toString() throws NullPointerException {
128     if (size == 0) {
129         return "[]";
130     }
131     Card currentCard = firstCard;
132     String result = "[";
133     while (currentCard.getNext() != null) {
134         result += currentCard + " ";
135         currentCard = (Card) currentCard.getNext();
136     }
137     if (getLastCard() != null) {
138         result += currentCard.toString();
```



```
139     } else {
140         throw new NullPointerException("The Deck is Empty");
141     }
142     //result += "["+currentCard.toString()+"]";
143     result += "]";
144     return result;
145 }
146
147 //string representation of the array
148 public String representBoard() throws NullPointerException {
149     String s = toString();
150     s += "\n";
151     String choice = "";
152     for (int i = 1; i < getSize() + 1; i++) {
153         if (i == 1) {
154             choice += " " + i;
155         } else {
156             choice += "   " + i;
157         }
158     }
159     return s + choice;
160 }
161
162 //grab the first card
163 public Card getFirstCard() throws NullPointerException {
164     if (firstCard == null)
165         throw new NullPointerException("None shall pass.");
166     return firstCard;
167 }
168
169 //grab the last card
170 public Card getLastCard() throws NullPointerException {
171     if (firstCard == null)
172         throw new NullPointerException("The Deck is Empty");
173     return lastCard;
174 }
175 //check answer of 00 format
176 public boolean checkAnswer(int pa, int pb) {
177     if (pa == pb) {
178         return false;
179     }
180     return checkAnswer(pa, pb, 0);
181 }
182
183 //overload for 000 format
184 public boolean checkAnswer(int pa, int pb, int pc) {
185     if (pa == pb || pa == pc || pb == pc) {
186         return false;
187     }
188     int a, b, c;
189     a = getNthCardValue(pa);
190     b = getNthCardValue(pb);
191     if (pc == 0) {
192         c = 0;
193     } else {
194         c = getNthCardValue(pc);
195     }
196
197     return ((a + b + c) == 36) || ((a + b) == 11);
198 }
199
200 //get the raw posish of the KJQ cards
201 private void getJQKPos() throws NullPointerException {
202     //return the raw posish's e.g. [7,8,9]
203     int[] posish = {0, 0, 0};
204
205     //iterate the board
206     for (int i = 1; i < size + 1; i++) {
207         int value = getNthCardValue(i);
208         //get the card point value on ith posish
```

```
209         switch (value) {
210             case 11 ->
211                 posish[0] = i;
212             case 12 ->
213                 posish[1] = i;
214             case 13 ->
215                 posish[2] = i;
216         }
217     }
218     if (posish[0] > 0 && posish[1] > 0 && posish[2] > 0) {
219         validMove.push(posish);
220     }
221
222 }
223 //is KJQ exists on the board
224 public boolean isJQK() {return cJ != 0 && cQ != 0 && cK != 0;}
225
226 public int getcJ() {return cJ;} //get the cJ counter
227
228 public int getcQ() {return cQ;} //get the cQ counter
229
230 public int getcK() {return cK;} //get the cK counter
231
232 public void clear() { //clear the board
233     firstCard.setNext(null);
234     lastCard.setNext(null);
235     size = 0;
236 } //clear a board (for tests)
237
238 public int getSize() {return size;} //get the size of the board
239
240 //check all valid moves except KJQ <- Board
241 public void searchValidMoves() {
242     //clear the stack to make sure
243     validMove.clear();
244     if (checkAnswer(1, 2) == true) {
245         int[] a = {1, 2};
246         validMove.push(a);
247     }
248
249
250     Card outerCard = firstCard;
251     int outer = 1;
252     //while while....
253     while (outerCard.getNext() != null) {
254         Card innerCard = (Card) firstCard.getNext();
255         Card offsetCard = innerCard;
256         int inner = getSize() - outer + 1;
257         while (innerCard.getNext() != null) {
258
259             innerCard = (Card) innerCard.getNext();
260             //System.out.println("i: " +innerCard+" | o: " +outerCard + "offset: " +
offsetCard);
261             if ((outerCard.getCardValue() + innerCard.getCardValue() == 11)) {
262                 int inner2 = size - outer;
263
264                 int[] valid = {0, 0};
265                 valid[0] = getCardPosition(innerCard);
266                 valid[1] = getCardPosition(outerCard);
267                 //push the found posish
268                 validMove.push(valid);
269             }
270
271             inner++;
272         }
273         outerCard = (Card) outerCard.getNext();
274         offsetCard = (Card) offsetCard.getNext();
275
276         outer++;
277     }
```

```
278         getJQKPos(); //filled up KJQ posish to stack, providing
279     }                //this element on the top of the stack to
280                     //maximizing efficiency
281
282
283     //update and return the valid move stack <-App.java
284     public Stack getValidMove() {
285         searchValidMoves();
286         return validMove;
287     }
288
289     //remove a specific card from the deck
290     public Card removeACard(Card aCard) throws CardNotFoundException {
291         //used at testing
292         Card c = (Card) firstCard;
293         Card tmpCard = c;
294         int i = 1;
295         boolean found = false;
296         //remove the first element
297         //of the list
298         if (c.getCardValue() == aCard.getCardValue() &&
299             c.getStrSuit() == aCard.getStrSuit()) {
300             firstCard = (Card) c.getNext();
301             size--;
302             found = true;
303         }
304         //iterate throughout the list
305         while (!found && c.getNext() != null) {
306             Card n = (Card) c.getNext();
307
308             //if both suit and value ar the same
309             //we found the card
310             if (n.getCardValue() == aCard.getCardValue() &&
311                 n.getStrSuit() == aCard.getStrSuit()) {
312                 found = true;
313                 size--;
314                 //skipp the next card
315                 c.setNext(n.getNext());
316                 tmpCard = n;
317             } else {
318                 //if no match, get the next card
319                 c = (Card) c.getNext();
320             }
321             i++;
322         } //endwhile
323         setlastcard();
324         if (!found)
325             throw new CardNotFoundException(aCard);
326         return tmpCard;
327     }
328
329     //used multiple times, so abstracted
330     private void setlastcard() {
331         //updates the card on the board
332         Card card = firstCard;
333         while (card.getNext() != null) {
334             card = (Card) card.getNext();
335         }
336         lastCard = card;
337     }
338
339     //remove a specific card number from the board
340     //used at testing
341     public Card removeNthCard(int num){
342         Card currentCard = (Card) firstCard;
343         Card tmpCard = currentCard;
344         //retrieve the very first card
345         if (num == 1) {
346             firstCard = (Card) currentCard.getNext();
347             size--;
```

```
348         return tmpCard;
349     }
350     int counter = 0;
351     //iterate throughout the board
352     while (counter <= num && currentCard.getNext() != null) {
353         Card nextCard = (Card) currentCard.getNext();
354
355         if (num - 2 == counter) {
356             size--;
357             currentCard.setNext(nextCard.getNext());
358             tmpCard = nextCard;
359
360         } else {
361             currentCard = (Card) currentCard.getNext();
362         }
363         counter++;
364     } //endwhile
365     setlastcard();
366     return tmpCard;
367 }
368
369 //get the Nth card point value <- Board.java
370 public int getNthCardValue(int num) throws NullPointerException {
371     if (num < 1 || num > size)
372         throw new NullPointerException("The selection is out of range");
373     Card currentCard = (Card) firstCard;
374     if (num == 1) return firstCard.getCardValue();
375     int counter = 0;
376     while (counter <= num && currentCard.getNext() != null) {
377         Card nextCard = (Card) currentCard.getNext();
378
379         if (num - 2 == counter) {
380             return nextCard.getCardValue();
381         } else {
382             currentCard = (Card) currentCard.getNext();
383         }
384         counter++;
385     } //endwhile
386     return -1;
387 }
388 //get the Nth card point value App uses this
389 public Card getNthCard(int num) throws NullPointerException {
390     if (num < 1 || num > size)
391         throw new NullPointerException("The selection is out of range");
392     Card currentCard = (Card) firstCard;
393     if (num == 1) return firstCard;
394     int counter = 0;
395     while (counter <= num && currentCard.getNext() != null) {
396         Card nextCard = (Card) currentCard.getNext();
397
398         if (num - 2 == counter) {
399             return nextCard;
400         } else {
401             currentCard = (Card) currentCard.getNext();
402         }
403         counter++;
404     } //endwhile
405     return new Card(0, "h");
406 }
407 }
```

CardInterFace.java

```
1 package arch;
2
3 public interface CardInterFace {
4     public Card removeFirstElement() throws IllegalStateException;
5     public boolean isEmpty();
6     public boolean addNewEntry(Object newEntry) throws LockedDeckException;
7     public Card[] toArray() throws EmptyDeckException;
8     public boolean addNewEntry(Card newEntry) throws LockedDeckException;
9     public Card getFirstCard() throws NullPointerException;
10    public Card getLastCard() throws NullPointerException;
11
12 }
```

Card.java

```
1 package arch;
2
3 public class Card extends Node{
4     private String suit, strSuit;
5     private boolean isFace;
6     private int cardValue;
7
8     //new Card(1,"h");
9     public Card(int pValue, String pSuit) {
10         super(pValue);
11         cardValue = validateValue(pValue);
12         setIsFaceCard();
13         validateSuit(pSuit);
14     }
15
16     public Card() throws NullPointerException {
17         throw new NullPointerException("Empty Card Cannot Be created");
18     }
19
20     //card point value restriction
21     private int validateValue(int value) throws IllegalStateException{
22         if (value >= 1 && value <= 13){
23             return value;
24         }else{
25             throw new IllegalStateException("Invalid CardValue: " + value);
26         }
27     }
28
29     //nice suit display
30     private void validateSuit(String pSuit) throws IllegalStateException{
31         String suitLower = pSuit.toLowerCase();
32         String tmpsuit = "";
33         if (suitLower.equals("s") || suitLower.equals("c") || suitLower.equals("h") || suitLower.equals("d")){
34             switch (suitLower){
35                 case "s":
36                     tmpsuit = "s"; //<---- changed due to latex error
37                     break;
38                 case "c":
39                     tmpsuit = "c";
40                     break;
41                 case "h":
42                     tmpsuit = "h";
43                     break;
44                 case "d":
45                     tmpsuit = "d";
46                     break;
47             }
48             strSuit = suitLower;
49             suit = tmpsuit;
50         }else{
51             throw new IllegalStateException("Invalid Suit: " + pSuit);
52         }
53     }
54     public String toString() throws IllegalStateException{ //representation
55         String tmpDenoted = "";
56         switch (cardValue) {
57             case 1: tmpDenoted = "A";
58             break;
59             case 2: case 3: case 4: case 5: case 6: case 7: case 8: case 9:
60                 tmpDenoted = Integer.toString(cardValue);
61                 break;
62             case 10: tmpDenoted = "T";
63             break;
64             case 11: tmpDenoted = "J";
65             break;
66             case 12: tmpDenoted = "Q";
67             break;
```

```
68         case 13: tmpDenoted = "K";
69             break;
70         default:
71             throw new IllegalStateException("Unknown cardValue: " + cardValue);
72     }
73     return tmpDenoted+this.suit;
74 }
75 public int getCardValue() {return cardValue;}           //get point value
76 e.g. J -> 11
77 protected void setCardValue(int v) {cardValue = v;}    //set a card value
78 e.g. 11 -> J
79 protected void setCardSuit(String s) {                 //stringify a suit
80     strSuit = s;
81     validateSuit(s);
82 }
83 public String getSuit()    {return suit;}               //format
84 public boolean isFace()    {return isFace;}             //is JQK?
85 public String getStrSuit(){return strSuit;}             //get the suit in
86 "h","s","p" format
87 private void setIsFaceCard(){isFace = cardValue >= 11 && cardValue <= 13;} //set faceCard
88 }
```

Deck.java

```
1 package arch;
2 import java.util.Random;
3 public class Deck<T> extends LinkedList implements CardInterFace {
4     private Card next, firstCard, lastCard;
5     private final int SHUFFLESIZE = 1000;
6     private boolean lock = false;
7     private static final String[] SUITS = {"h","d","s","c"};
8
9     //build a standard deck
10    public Deck() throws LockedDeckException, NullPointerException{
11        for (int r = 1; r <= 13 ; r++) {
12            for (int s = 0; s <= 3 ; s++) {
13                addNewEntry(new Card(r, SUITS[s]));
14            }
15        }
16    }
17    public Deck(boolean shuffle, boolean fill)throws LockedDeckException,
18        NullPointerException, EmptyDeckException{
19        //optional shuffle and fill
20        if (fill) {
21            for (int r = 1; r <= 13; r++) {
22                for (int s = 0; s <= 3; s++) {
23                    addNewEntry(new Card(r, SUITS[s]));
24                }
25            }
26        }
27        if (shuffle){
28            shuffle();
29            lock =true;
30        }
31
32    }
33    public void shuffle() throws LockedDeckException, EmptyDeckException {
34        //deck is locked after shuffle, disabling more shuffle
35        if (lock == true)
36            throw new LockedDeckException("Deck is locked");
37        //empty deck cannot be shuffled
38        if (size == 0)
39            throw new EmptyDeckException("The Deck is Empty");
40        Random random = new Random();
41        //shuffle N times
42        for (int i = 0; i < size*SHUFFLESIZE; i++) {
43            int randomElement = random.nextInt(size - 1);
44            int j = 0;
45            Card currentCard = firstCard;
46            //iterate to the card
47            while (j < randomElement && currentCard.getNext() != null) {
48                currentCard = (Card) currentCard.getNext();
49                j++;
50            }
51            //insert it to the front
52            Card first = (Card) currentCard;
53            Card SecondCard = (Card) currentCard.getNext();
54            Card thirdCard = (Card) currentCard.getNext().getNext();
55
56            first.setNext(thirdCard);
57            SecondCard.setNext(this.firstCard);
58            this.firstCard = SecondCard;
59            //if we pick the last element set size-2 card to last
60            if (randomElement == size - 2){
61                this.lastCard = currentCard;
62            }
63        }
64        //lock the deck to avoid shuffle
65        lock = true;
66    }
67    public Card removeFirstElement() throws IllegalStateException{
68        if (size == 0){throw new IllegalStateException("Cannot " +
```



```
69         "remove a Card from an empty deck");}
70     //removing the very first card
71     if(firstCard != null){
72         Card first = (Card) firstCard;
73         firstCard = (Card) firstCard.getNext();
74         size--;
75         //for integrity
76         if (size == 1) updateLastElement();
77         //if card removed return with it
78         return first;
79         //return a BIG null
80     }else return null;
81 }
82
83 //remove a specific card from the deck
84 public void removeFirstElement(Card aCard) throws CardNotFoundException{
85     Card currentCard = (Card) firstCard;
86     boolean found = false;
87     //remove the first element
88     if (currentCard.getCardValue() == aCard.getCardValue() &&
89         currentCard.getStrSuit() == aCard.getStrSuit()) {
90         firstCard = (Card) currentCard.getNext();
91         size--;
92         found = true;
93     }
94     while(!found && currentCard.getNext() != null){
95         Card nextCard = (Card) currentCard.getNext();
96
97         //if both suit and value are the same //we found the card
98         if (nextCard.getCardValue() == aCard.getCardValue() &&
99             nextCard.getStrSuit() == aCard.getStrSuit()) {
100             found = true;
101             size--;
102             //skipp the next card
103             currentCard.setNext(nextCard.getNext());
104         }else{
105             //if no match, get the next card
106             currentCard = (Card) currentCard.getNext();
107         }
108     }//endwhile
109     setlastcard();
110     if (!found)
111         throw new CardNotFoundException(aCard);
112 }
113
114 //update the last card safety
115 public void setlastcard(){
116     Card card = firstCard;
117     while(card.getNext() != null) {
118         card = (Card) card.getNext();
119     }
120     lastCard = card;
121 }
122
123 //add a new card to the deck
124 public boolean addNewEntry(Card newEntry) throws LockedDeckException{
125     //when the deck is locked (after shuffle), disable the method
126     if (lock){
127         throw new LockedDeckException("Deck is Locked, not modifiable");
128     }
129     Card newCard = newEntry;
130     newCard.setNext(firstCard);
131     firstCard = newCard;
132     size++;
133     //if we have only one node, we record it as a last node
134     if (getSize() == 1)
135         lastCard = newCard;
136     return true;
137 }
138
```

```
139 //array representation of the deck
140 public Card[] toArray() throws EmptyDeckException{
141     Card[] cardArray = new Card[size];
142     Card[] emptyArray = new Card[0];
143
144     if (isEmpty() == true)
145         throw new EmptyDeckException("Deck is empty");
146
147     int counter = 0;
148     Card currentCard = firstCard;
149     while (currentCard.getNext() != null) {
150         cardArray[counter] = currentCard;
151         currentCard = (Card) currentCard.getNext();
152         counter++;
153     }
154     return cardArray;
155 }
156 public int getSize(){ //get the size of the deck
157     return size;
158 } //return the deck size
159
160 //string representation of the array
161 public String toString() throws NullPointerException{
162     Card currentCard = firstCard;
163     String result = "[";
164     //fill the array
165     while (currentCard.getNext() != null) {
166         result += currentCard.toString()+" ";
167         currentCard = (Card) currentCard.getNext();
168     }
169     if (getLastCard() != null){
170         result += currentCard.toString();
171     }else{
172         throw new NullPointerException("The Deck is Empty");
173     }
174     //result += "["+currentCard.toString()+"]";
175     result += "]";
176     return result;
177 }
178
179 //grab the first card
180 public Card getFirstCard() throws NullPointerException {
181     if (firstCard== null)
182         throw new NullPointerException("The Deck is Empty");
183     return firstCard;
184 }
185
186 //grab the last card
187 public Card getLastCard() throws NullPointerException {
188     if (firstCard== null)
189         throw new NullPointerException("The Deck is Empty");
190     return lastCard;
191 }
192 }
```

LinkedList.java

```
1 package arch;
2
3 public class LinkedList<T> {
4     protected Node<T> firstNode;
5     protected Node<T> lastNode;
6     protected Integer size;
7
8     public LinkedList() {
9         firstNode = null;
10        lastNode = null;
11        size = 0;
12    }
13
14    public Node<T> getFirstNode() {
15        return firstNode;
16    } //return first node
17
18    public Node<T> getLastNode() {
19        return lastNode;
20    } //return last node
21
22    public int getSize() {
23        return size;
24    } //get size
25
26    public boolean isEmpty() {
27        return size == 0;
28    } //is empty
29
30    protected void updateLastElement() {
31        //this method updates the last node, which is required
32        // if we remove exactly the last element from the list
33        if (getSize() == 0) {
34            lastNode = null;
35        } else if (getSize() == 1) {
36            lastNode = firstNode;
37        } else {
38            Node<T> currentNode = firstNode;
39            while (currentNode != null) {
40                if (currentNode.getNext() != null) {
41                    currentNode = currentNode.getNext();
42                } else {
43                    lastNode = currentNode;
44                    break;
45                }
46            }
47        }
48    }
49
50    //add new element to the data structure
51    public boolean addNewEntry(Object newEntry) {
52        Node<T> newNode = new Node<T>((T) newEntry);
53        newNode.setNext(firstNode);
54        firstNode = newNode;
55        size++;
56        //if we have only one node, we record it as a last node
57        if (getSize() == 1)
58            lastNode = newNode;
59        return true;
60    }
61
62    //remove the first element without getting the object
63    public T removeFirstElement() throws NullPointerException{
64        if(size==0){
65            throw new NullPointerException("Empty list");
66        }
67        if (firstNode != null) {
68            T result = firstNode.getData();
```

```
69         firstNode = firstNode.getNext();
70         size--;
71         updateLastElement();
72         return result;
73     } else return null;
74 }
75
76 //remove first, return with first
77 public boolean removeSpfecific(Object anEntry) {
78     Node<T> nodeToRemove = findEntry((T) anEntry);
79     if (nodeToRemove.getData() == null) {
80         return false;
81     }
82     Node<T> node = firstNode;
83
84     nodeToRemove.setData(firstNode.getData());
85     firstNode = firstNode.getNext();
86     size--;
87     return true;
88 }
89
90 //clear everything
91 protected void clear() {
92     firstNode.setNext(null);
93     lastNode.setNext(null);
94     size = 0;
95 }
96
97 //find a given entry
98 protected Node<T> findEntry(T nodeToFind) {
99     Node<T> currentNode = firstNode;
100     boolean found = false;
101
102     while (!found && currentNode.getNext() != null) {
103         if (currentNode.getData().equals(nodeToFind)) {
104             found = true;
105             return currentNode;
106         } else {
107             currentNode = currentNode.getNext();
108         }
109     }
110     if (lastNode.getData() == nodeToFind) {
111         return lastNode;
112     }
113     return null;
114 }
115
116 //representation in array
117 public T[] toArray() throws EmptyDeckException {
118     T[] resultArray = (T[]) new Object[size];
119
120     if (isEmpty() == true)
121         return resultArray;
122
123     int counter = 0;
124     Node<T> currentNode = firstNode;
125
126     while (currentNode.getNext() != null) {
127         resultArray[counter] = currentNode.getData();
128         currentNode = currentNode.getNext();
129         counter++;
130     }
131     return resultArray;
132 }
133
134 //string representation
135 public String toString() {
136     if (isEmpty() == true) {
137         return "[]";
138     }
```

```
139     Node<T> currentNode = firstNode;
140     String result = "[";
141     while (currentNode.getNext() != null) {
142         result += currentNode.getData();
143         currentNode = currentNode.getNext();
144     }
145     if (getLastNode() != null)
146         result += lastNode.getData();
147     result += "]";
148     return result;
149 }
150 }
```

Node.java

```
1 package arch;
2 //setting up individual nodes, which can point to another element
3 public class Node<T> {
4     private T cardValue;
5     private Node<T> next;
6
7     public Node() throws NullPointerException{
8         throw new NullPointerException("No data in node!");
9     }
10
11     public Node(T dataValue) {
12         cardValue = dataValue;
13         next = null;
14     }
15
16     //get the value of the node
17     public T getData() {
18         return cardValue;
19     }
20
21     //set the node data
22     public void setData(T dataValue) {
23         cardValue = dataValue;
24     }
25
26     //get the next node
27     public Node<T> getNext() {
28         return next;
29     }
30
31     //set the next node
32     public void setNext(Node<T> nextNode) {
33         next = nextNode;
34     }
35
36     //string repr
37     public String toString(){
38         if (getData() != null) {
39             return "Data:\t" + getData()+ "\t->\t" + getNext();
40         }else{
41             return "Data:\t null";
42         }
43     }
44 }
```

ReplayItem.java

```
1 package arch;
2
3 public class ReplayItem extends Node{
4     private String data;
5
6     public ReplayItem(String pValue) {
7         super(pValue);
8         data = pValue;
9     }
10
11     public String toString(){
12         return data;
13     }
14
15     public String getData() {return data;}
16 }
```

Replay.java

```
1 package arch;
2
3 import java.util.Scanner;
4
5 public class Replay<T> extends Stack implements StackInterface {
6     private ReplayItem firstItem, lastItem;
7
8     //add a new item to replay
9     public boolean push(ReplayItem pItem) throws LockedDeckException {
10         if (firstItem != null) {
11             pItem.setNext(firstItem);
12         }
13         firstItem = pItem;
14
15         size++;
16         //if we have only one node, we record it as a last node
17         if (size == 1)
18             lastItem = pItem;
19         return true;
20     }
21
22     //array representation of the replay
23     public ReplayItem[] toArray() throws EmptyDeckException {
24         ReplayItem[] itemArray = new ReplayItem[size];
25         ReplayItem[] emptyArray = new ReplayItem[0];
26
27         if (isEmpty() == true)
28             throw new EmptyDeckException("Deck is empty");
29
30         //build the result array
31         int counter = 0;
32         ReplayItem currentItem = firstItem;
33         while (currentItem.getNext() != null) {
34             itemArray[counter] = currentItem;
35             currentItem = (ReplayItem) currentItem.getNext();
36             counter++;
37         }
38         return itemArray;
39     }
40
41     //get the size of the replay
42     public int getSize() {
43         return size;
44     }
45
46     //string representation
47     public String toString() throws NullPointerException {
48
49         ReplayItem currentItem = firstItem;
50         String result = "";
51         //result += firstItem.getData()+"\n";
52         ReplayItem nextItem = (ReplayItem) currentItem.getNext();
53
54         //fill the array
55         while (currentItem.getNext() != null) {
56             result = currentItem.getData();
57             currentItem = (ReplayItem) currentItem.getNext();
58             System.out.println(result);
59
60         }
61         System.out.println(lastItem.getData());
62         result = lastItem.getData() + result + "\n";
63
64         return "";
65     }
66
67     //grab the last replay item
68     public ReplayItem getLastItem() throws NullPointerException {
```



```
69         if (firstItem == null)
70             throw new NullPointerException("The Deck is Empty");
71         return lastItem;
72     }
73
74     //grab the last card and delete it
75     public String getLastItemAndDelete(){
76         if (size >0){
77             ReplayItem item = getLastItem();
78
79         }
80         String tmp = "-1";
81         ReplayItem currentItem = firstItem;
82         for (int i = 0; i < size-1; i++) {
83             currentItem = (ReplayItem) currentItem.getNext();
84         }
85         tmp = currentItem.getData();
86         currentItem.setNext(null);
87         size--;
88         return tmp;
89     }
90     //print out in reverse
91     public void printHistory(){
92         Scanner scan = new Scanner(System.in);
93         while(size != 0){
94             System.out.println(getLastItemAndDelete());
95             System.out.print(">>>");
96             scan.nextLine();
97         }
98         System.out.println(">>>Bye...");
99     }
100 }
```

StackInterface.java

```
1 package arch;
2
3 public interface StackInterface {
4     public int[] peek();
5     public int getSize();
6     public boolean push(int[] newEntry);
7     public boolean isEmpty();
8     public int[] pop ();
9     public void clear ();
10 }
```

Stack.java

```
1 package arch;
2
3 public class Stack<T> implements StackInterface{
4     protected Node<T> firstNode;
5     protected Node<T> lastNode;
6     protected int[] data;
7     protected int size;
8
9     public Stack() {
10         firstNode = null;
11         size = 0;
12     }
13
14     public int[] peek() {
15         return (int[]) firstNode.getData();
16     }
17
18     public int getSize() {
19         return size;
20     }
21
22     public boolean isEmpty() {
23         return size == 0;
24     }
25
26     public boolean push(int[] newEntry) {
27         Node<T> newNode = new Node(newEntry);
28         newNode.setNext(firstNode);
29         int[] arr = newEntry;
30         for (int i = 0; i < arr.length; i++) {
31             for (int j = i + 1; j < arr.length; j++) {
32                 int tmp = 0;
33                 if (arr[i] < arr[j]) {
34                     tmp = arr[i];
35                     arr[i] = arr[j];
36                     arr[j] = tmp;
37                 }
38             }
39         }
40         newNode.setData((T) arr);
41         firstNode = newNode;
42         size++;
43         if (size == 1) {
44             lastNode = newNode;
45         }
46         return true;
47     }
48
49     public int[] pop () {
50         if (firstNode != null) {
51             int[] result = (int[]) firstNode.getData();
52             firstNode = firstNode.getNext();
53             size--;
54             return result;
55         } else return null;
56     }
57
58     public void clear () {
59         if (firstNode != null) {
60             firstNode.setNext(null);
61             size = 0;
62         }
63     }
64
65     public String toString () {
66         if (isEmpty()) {
67             return "[]";
68         }
```

```
69         String result = "[";
70         Node<T> currentNode = firstNode;
71
72         while (currentNode.getNext() != null) {
73             for (int i : (int[]) currentNode.getData()) {
74                 result += i;
75             }
76             result += "|";
77             currentNode = currentNode.getNext();
78
79         }
80         for (int i : (int[]) lastNode.getData()) {
81             result += i;
82         }
83         result += "]";
84         return result;
85     }
86 }
```

This is pdfTeX, Version 3.141592653-2.6-1.40.22 (TeX Live 2021/Arch Linux) kpathsea version 6.3.3