Ulster
University

# Java Source Code - Elevens

**Author**

Mr. Adam Torok - B00798824

Jordanstown, April 2021

## App.java

```java
package arch;

import java.util.Scanner;

public class App {
    Scanner scan = new Scanner(System.in);
    private int gameState = 0; //0 running game | 1 win | -1 staleMate
    private Replay replay = new Replay();


    public App() throws LockedDeckException, EmptyDeckException {}

    public void selectMode() throws LockedDeckException, EmptyDeckException { //select and run the
     mode
        int user;

        System.out.println("CardGame\n-----------------");
        System.out.println("1. Play\n2. Demonstration Mode\n3. Rules\n4. Exit");
        Scanner scan = new Scanner(System.in);//this is for testing
        user = Integer.parseInt(scan.nextLine());

        //until we get a valid choice
        //call the corresponding mode
        while (user != 4) {
            if (user == 1) {
                System.out.println("Play Mode\n");
                playMode(false);
            } else if (user == 2) {
                System.out.println("Demonstration Mode");
                playMode(true);
            } else if (user == 3) {
                System.out.println("The Rules of the Game\n");
                showRules();
            } else if (user == 4) {
                System.out.println("Bye...");
            }
        }
    }

    //main game
    public void playMode(boolean boo) throws LockedDeckException, EmptyDeckException {
        System.out.println("Dealing Your cards......");
        //get a new deck and board
        Deck d = new Deck(true, true);
        Board b = new Board();
        //deal the BOARDSIZE amount of cards
        for (int i = 0; i < b.BOARDSIZE; i++) {
            b.addNewEntry(d.removeFirstElement());
        }
        //print out the board
        System.out.println(b.representBoard());
        System.out.println("----------------------------------------");
        String initBoard = "Initial board: " + b.toString();
        ReplayItem r = new ReplayItem(initBoard);
        replay.push(r);

        //until no valid moves or (deck and board
        while (gameState != -1 || (d.getSize() == 0 && b.getSize() == 0)) {
            //gets empty
            if (gameState == -1) {
                break;                  //break out
            }
            //clear the valid moves stack
            b.getValidMove().clear();
            int cardRemoved = 0;
            boolean acceptableMove = false;

            //get all valid moves
            Stack v = b.getValidMove();
            int[] choice = {0, 0, 0};
            //if the valid stack has moves
            if (v.getSize() > 0) {
                if (boo == true) {
                    //the machine will auto peek to learn a posish
                    choice = v.peek();
                } else { //USER MODE
                    //user interface, hint, validate
                    boolean proceed = false;
                    while (proceed == false) {
                        System.out.print("[" + d.getSize() + "]]]]\tPlease Select Card: ");
                        String user = scan.nextLine();
                        if (user.length() > 0 && user.length() <= 3 && !user.equals("h")) {
                            for (int i = 0; i <= user.length() - 1; i++) {
                                choice[i] = user.charAt(i) - 48;
                            }
                            //order the users input
                            int[] tmpArr = choice;
                            for (int i = 0; i <= tmpArr.length; i++) {
                                for (int j = i + 1; j < tmpArr.length; j++) {
                                    int tmp = 0;
                                    if (tmpArr[i] < tmpArr[j]) {
                                        tmp = tmpArr[i];
                                        tmpArr[i] = tmpArr[j];
                                        tmpArr[j] = tmp;
                                    }
                                }
                            }
                            //validate their selection
                            if (!b.checkAnswer(choice[0], choice[1]) &&
                                    b.checkAnswer(choice[0], choice[1], choice[2]) == false) {
                                System.out.println("This is not good selection");
                            } else {
                                //invalid input will stop and ask again
                                proceed = true;
                                break;
                            }
                            //activate hint & auto remove
                        } else if (user.equals("h")) {
                            choice = v.peek();
                            String hint = "";
                            for (int i = 0; i < choice.length; i++) {
                                hint += Integer.toString(choice[i]) + "  ";
                            }
                            System.out.println("You can continue like this -> " + hint);
                            System.out.println("Let me help ya, I remove it for you...");
                            proceed = true;
                            break;
                        }
                    }
                }
                //extra security, validating the move on
                //the board as well | input nn0
                if (choice.length == 2 || choice[2] == 0) {
                    cardRemoved = 2;
                    acceptableMove = b.checkAnswer(choice[0], choice[1]);
                //testing input nnn
                } else if (choice.length == 3) {
                    cardRemoved = 3;
                    acceptableMove = b.checkAnswer(choice[0], choice[1], choice[2]);
                } else {
                    gameState = -1;
                    break;
                }
            //no valid moves in the stack so stalemate
            } else {
                gameState = -1;
                System.out.println("No valid Moves");
                break;
            }
            //if the users move is acceptable
            if (acceptableMove) {
                //prepare the input for replay
                String choosen = "";
                choosen += b.getNthCard(choice[0]).toString() + "   ";
                choosen += b.getNthCard(choice[1]).toString() + "   ";
                if (choice.length == 3 && choice[2] != 0) {
                    choosen += b.getNthCard(choice[2]).toString() + "   ";
                }
                //reprint the selection
                System.out.println("Removed Cards:\t" + choosen);

                //solve the KJQ tab issue
                if (b.getSize() > 0) {
                    if (choice.length == 3) {
                        choosen += "\tafter-->\t" + b.toString();
                    } else {
                        choosen += "\t\tafter-->\t" + b.toString();
                    }
                }

                //push the selection to the stack
                ReplayItem rinit = new ReplayItem(choosen);
                replay.push(rinit);

                if (b.getSize() == 2) {
                    //artificial board clear when two cards left
                    //anytime it happens, that is a won scen.
                    // to avoid mess up with the replay
                    b.clear();
                } else {
                    for (int i : choice) {
                        b.removeNthCard(i);
                    }
                }
                //if deck has 2 card we cant remove 3
                if (d.getSize() > 0) {
                    int toReplace = Math.min(d.getSize(), cardRemoved);
                //replace the cards
                    for (int i = 0; i < toReplace; i++) {
                        b.addNewEntry(d.removeFirstElement());
                    }
                }
                //no deck, no board means a win
                if (d.getSize() == 0 && b.getSize() == 0) {
                    gameState = 1;
                    break;
                }
                //display the empty board
                System.out.println("-----------------");
                System.out.println(b.representBoard());
            }
        }
        //things relating to the game state
        switch (gameState) {
            //the game is over
            case -1:
                System.out.println("\n\"Game over man, GAME OVER!\" - Pvt. Hudson");
                System.out.println("Cards Left: " + d.getSize());
                break;
            //win
            //create a new board to proper cleanup
            case 1:
                System.out.println("\nHold my beer...... YOU WON\n\n");
                b = new Board();
        }
        //handling replay and quit
        System.out.println("\nWould you like to see the replay? (y/n) - (q) quit");
        String choice = scan.next();
        //render replay
        if (choice.equals("y") || choice.equals("Y")) {
            String finalBoard = "Final Board:\t" + b.toString();

            System.out.println("Action Replay!");
            System.out.println("Replay Size: " + replay.size);
            replay.push(new ReplayItem(finalBoard));
            replay.printHistory();
        //quit
        } else if (choice.equals("q") || choice.equals("Q")
                || choice.equals("n") || choice.equals("N")) {
            System.exit(0);
        }
    }
    //nice touch!
    public static void showRules() {
        System.out.println("Elevens is extremely similar to Bowling Solitaire,\n" +
                "except that the layout is a little different and\n" +
                "the goal is to make matching pairs that add up to\n" +
                "11 rather than adding matching pairs up to 10.\n" +
                "\n" +
                "Empty spaces in the 9-card formation are automatically\n" +
                "filled by placing a card from the Deck in the free space.\n" +
                "Once you run out of cards in the Deck, do not fill the\n" +
                "empty spaces in the card formation with any other cards.\n" +
                "\n" +
                "To play this game, look at your 9-card formation and see\n" +
                "if any cards can be matched that add up to 11 in total.\n" +
                "If you have a matching pair that can create this sum, \n" +
                "then you may remove them from place. Once youve done so,\n" +
                "remember to fill in the gaps left by these two cards with\n" +
                "two cards from the Deck.\n" +
                "\n" +
                "Only cards in the 9-card formation are available to play\n" +
                "with, and you may not build any cards on top of each other\n" +
                "during the game. Cards cannot be removed from the Deck \n" +
                "unless they are being placed in the table layout, and you \n" +
                "should not look at the cards in the Deck before moving them\n" +
                "into play. They must remain unknown until they are flipped\n" +
                "over to be placed in the 9-card formation.\n" +
                "\n" +
                "The ranking of cards matches their face value i.e. the two of\n" +
                "clubs is equal to two. Aces hold a value of one and Jacks, \n" +
                "Queens, and Kings equal eleven only when they are removed \n" +
                "together. For example, if you have a Jack and King on your \n" +
                "board you cant remove either until a Queen appears. Once all\n" +
                "three cards are present on the board they can be removed \n" +
                "together to make 11. They are the only cards in the game \n" +
                "that are moved as a trio, rather than being matched as a pair.\n" +
                "\n" +
                "HOW TO WIN:\n" +
                "\n" +
                "To win at a round of Elevens, you must remove absolutely all\n" +
                "cards from play  including those from the Deck. Once you \n" +
                "have matched all cards in the Deck, then you have won the round.\n" +
                "\n" +
                "It is possible to play this game with more than one player. \n" +
                "To do so, you could create a scoring system by having each \n" +
                "player keep their matched pairs and making each set worth 1 point.\n" +
                "The player with the highest number of points would win the game.\n" +
                "Typically, this is a solo player game, but its extremely easy to\n" +
                "make into a family-friendly or party game.");
    }

    public static void main(String[] args) throws LockedDeckException, EmptyDeckException {
        App a = new App();
        try {
            a.selectMode();
        } catch (NumberFormatException e) {
            System.out.println("Invalid Output! Lets try that again");
            a.selectMode();
        } catch (LockedDeckException e) {
            e.printStackTrace();
        } catch (EmptyDeckException e) {
            e.printStackTrace();
        }
    }
}
```

## Board.java

```java
package arch;

public class Board extends LinkedList<Card> implements CardInterFace {
    private Card firstCard, lastCard;
    private Stack validMove = new Stack();
    int cJ = 0, cQ = 0, cK = 0;
    public final int BOARDSIZE = 9;

    //just create an empty deck
    public Board(boolean b) throws LockedDeckException, NullPointerException {
        firstCard = null;
        lastCard = null;
    }

    public Board() {
        firstCard = null;
        lastCard = null;
    }

    //add a new card to the board
    public boolean addNewEntry(Card newEntry) {
        Card newCard = newEntry;
        newCard.setNext(firstCard);
        firstCard = newCard;
        size++;

        //if we have only one node, we record it as a lastCard
        if (getSize() == 1) {
            lastCard = newCard;
            lastCard.setNext(null);
        }
        //update KJQ counter
        switch (newEntry.getCardValue()) {
            case 11:
                cJ++;
                break;
            case 12:
                cQ++;
                break;
            case 13:
                cK++;
                break;
        }
        sort();
        return true;
    }

    public void sort() {
        if (size > 2) {
            //bubble sort the board
            for (int i = 0; i < size - 1; i++) {
                Card currentCard = firstCard;
                Card nextCard = (Card) firstCard.getNext();
                Card prevCard = null;
                for (int j = 1; j < size; j++) {
                    if (j == 1) {
                        prevCard = firstCard;
                    }
                    //if the current card value is greater
                    if (currentCard.getCardValue() > nextCard.getCardValue()) {
                        String tmpSuit = currentCard.getStrSuit();
                        int tmpValue = currentCard.getCardValue();
                        //swap the cards
                        currentCard.setCardValue(nextCard.getCardValue());
                        currentCard.setCardSuit(nextCard.getStrSuit());
                        nextCard.setCardValue(tmpValue);
                        nextCard.setCardSuit(tmpSuit);
                        //currentCard.setNext(nextCard.getNext());
                    }

                    //slipping the prevCard
                    //slipping the current and nexCards
                    prevCard = currentCard;
                    currentCard = nextCard;
                    nextCard = (Card) nextCard.getNext();

                }
            }
        }
    }

    //get an exact cards position
    public int getCardPosition(Card c) {
        int i = 1;
        Card currentCard = firstCard;
        while (currentCard.getNext() != null) {

            if (currentCard.getCardValue() == c.getCardValue() &&
                    currentCard.getStrSuit() == c.getStrSuit()) {
                return i;
            }

            if (currentCard.getCardValue() == c.getCardValue() &&
                    "*" == c.getStrSuit()) {
                return i;
            }

            currentCard = (Card) currentCard.getNext();
            i++;
        }
        if (lastCard.getCardValue() == c.getCardValue() &&
                lastCard.getStrSuit() == c.getStrSuit()) {
            return i++;
        }
        return -1;
    }
    //array representation of the board
    public Card[] toArray() throws NullPointerException {
        Card[] cardArray = new Card[size];
        Card[] emptyArray = new Card[0];

        if (isEmpty() == true)
            throw new NullPointerException("Board is empty");

        int counter = 0;
        Card currentCard = firstCard;
        while (currentCard.getNext() != null) {
            cardArray[counter] = currentCard;
            currentCard = (Card) currentCard.getNext();
            counter++;
        }
        //cardArray[size-1] = (Card) getLastCard();
        return cardArray;
    }

    //string representation of the array
    public String toString() throws NullPointerException {
        if (size == 0) {
            return "[]";
        }
        Card currentCard = firstCard;
        String result = "[";
        while (currentCard.getNext() != null) {
            result += currentCard + "  ";
            currentCard = (Card) currentCard.getNext();
        }
        if (getLastCard() != null) {
            result += currentCard.toString();
```

```java
        } else {
            throw new NullPointerException("The Deck is Empty");
        }
        //result += "["+currentCard.toString()+"]";
        result += "]";
        return result;
    }

    //string representation of the array
    public String representBoard() throws NullPointerException {
        String s = toString();
        s += "\n";
        String choice = "";
        for (int i = 1; i < getSize() + 1; i++) {
            if (i == 1) {
                choice += " " + i;
            } else {
                choice += "   " + i;
            }
        }
        return s + choice;
    }

    //grab the first card
    public Card getFirstCard() throws NullPointerException {
        if (firstCard == null)
            throw new NullPointerException("None shall pass.");
        return firstCard;
    }

    //grab the last card
    public Card getLastCard() throws NullPointerException {
        if (firstCard == null)
            throw new NullPointerException("The Deck is Empty");
        return lastCard;
    }
    //check answer of 00 format
    public boolean checkAnswer(int pa, int pb) {
        if (pa == pb) {
            return false;
        }
        return checkAnswer(pa, pb, 0);
    }

    //overload for 000 format
    public boolean checkAnswer(int pa, int pb, int pc) {
        if (pa == pb || pa == pc || pb == pc) {
            return false;
        }
        int a, b, c;
        a = getNthCardValue(pa);
        b = getNthCardValue(pb);
        if (pc == 0) {
            c = 0;
        } else {
            c = getNthCardValue(pc);
        }

        return ((a + b + c) == 36) || ((a + b) == 11);
    }

    //get the raw posish of the KJQ cards
    private void getJQKPos() throws NullPointerException {
        //return the raw posish's e.g. [7,8,9]
        int[] posish = {0, 0, 0};

        //iterate the board
        for (int i = 1; i < size + 1; i++) {
            int value = getNthCardValue(i);
            //get the card point value on ith posish
            switch (value) {
                case 11 ->
                        posish[0] = i;
                case 12 ->
                        posish[1] = i;
                case 13 ->
                        posish[2] = i;
            }
        }
        if (posish[0] > 0 && posish[1] > 0 && posish[2] > 0) {
            validMove.push(posish);
        }

    }
    //is KJQ exists on the board
    public boolean isJQK() {return cJ != 0 && cQ != 0 && cK != 0;}

    public int getcJ() {return cJ;}  //get the cJ counter

    public int getcQ() {return cQ;}  //get the cQ counter

    public int getcK() {return cK;}  //get the cK counter

    public void clear() {                                              //clear the board
        firstCard.setNext(null);
        lastCard.setNext(null);
        size = 0;
    }      //clear a board (for tests)

    public int getSize() {return size;}  //get the size of the board

    //check all valid moves except KJQ <- Board
    public void searchValidMoves() {
        //clear the stack to make sure
        validMove.clear();
        if (checkAnswer(1, 2) == true) {
            int[] a = {1, 2};
            validMove.push(a);
        }


        Card outerCard = firstCard;
        int outer = 1;
        //while while....
        while (outerCard.getNext() != null) {
            Card innerCard = (Card) firstCard.getNext();
            Card offsetCard = innerCard;
            int inner = getSize() - outer + 1;
            while (innerCard.getNext() != null) {

                innerCard = (Card) innerCard.getNext();
                //System.out.println("i: " +innerCard+" | o: " +outerCard + "offset: " +
    offsetCard);
                if ((outerCard.getCardValue() + innerCard.getCardValue() == 11)) {
                    int inner2 = size - outer;

                    int[] valid = {0, 0};
                    valid[0] = getCardPosition(innerCard);
                    valid[1] = getCardPosition(outerCard);
                    //push the found posish
                    validMove.push(valid);
                }

                inner++;
            }
            outerCard = (Card) outerCard.getNext();
            offsetCard = (Card) offsetCard.getNext();

            outer++;
        }
        getJQKPos(); //filled up KJQ posish to stack, providing
    }                //this element on the top of the stack to
                     //maximizing efficiency


    //update and return the valid move stack <-App.java
    public Stack getValidMove() {
        searchValidMoves();
        return validMove;
    }

    //remove a specific card from the deck
    public Card removeACard(Card aCard) throws CardNotFoundException {
        //used at testing
        Card c = (Card) firstCard;
        Card tmpCard = c;
        int i = 1;
        boolean found = false;
        //remove the first element
        //of the list
        if (c.getCardValue() == aCard.getCardValue() &&
                c.getStrSuit() == aCard.getStrSuit()) {
            firstCard = (Card) c.getNext();
            size--;
            found = true;
        }
        //iterate throughout the list
        while (!found && c.getNext() != null) {
            Card n = (Card) c.getNext();

            //if both suit and value ar the same
            //we found the card
            if (n.getCardValue() == aCard.getCardValue() &&
                    n.getStrSuit() == aCard.getStrSuit()) {
                found = true;
                size--;
                //skipp the next card
                c.setNext(n.getNext());
                tmpCard = n;
            } else {
                //if no match, get the next card
                c = (Card) c.getNext();
            }
            i++;
        }//endwhile
        setlastcard();
        if (!found)
            throw new CardNotFoundException(aCard);
        return tmpCard;
    }

    //used multiple times, so abstracted
    private void setlastcard() {
        //updates the card on the board
        Card card = firstCard;
        while (card.getNext() != null) {
            card = (Card) card.getNext();
        }
        lastCard = card;
    }

    //remove a specific card number from the board
    //used at testing
    public Card removeNthCard(int num){
        Card currentCard = (Card) firstCard;
        Card tmpCard = currentCard;
        //retrieve the very first card
        if (num == 1) {
            firstCard = (Card) currentCard.getNext();
            size--;
            return tmpCard;
        }
        int counter = 0;
        //iterate throughout the board
        while (counter <= num && currentCard.getNext() != null) {
            Card nextCard = (Card) currentCard.getNext();

            if (num - 2 == counter) {
                size--;
                currentCard.setNext(nextCard.getNext());
                tmpCard = nextCard;

            } else {
                currentCard = (Card) currentCard.getNext();
            }
            counter++;
        }//endwhile
        setlastcard();
        return tmpCard;
    }

    //get the Nth card point value <- Board.java
    public int getNthCardValue(int num) throws NullPointerException {
        if (num < 1 || num > size)
            throw new NullPointerException("The selection is out of range");
        Card currentCard = (Card) firstCard;
        if (num == 1) return firstCard.getCardValue();
        int counter = 0;
        while (counter <= num && currentCard.getNext() != null) {
            Card nextCard = (Card) currentCard.getNext();

            if (num - 2 == counter) {
                return nextCard.getCardValue();
            } else {
                currentCard = (Card) currentCard.getNext();
            }
            counter++;
        }//endwhile
        return -1;
    }
    //get the Nth card point value App uses this
    public Card getNthCard(int num) throws NullPointerException {
        if (num < 1 || num > size)
            throw new NullPointerException("The selection is out of range");
        Card currentCard = (Card) firstCard;
        if (num == 1) return firstCard;
        int counter = 0;
        while (counter <= num && currentCard.getNext() != null) {
            Card nextCard = (Card) currentCard.getNext();

            if (num - 2 == counter) {
                return nextCard;
            } else {
                currentCard = (Card) currentCard.getNext();
            }
            counter++;
        }//endwhile
        return new Card(0, "h");
    }
}
```

## CardInterFace.java

```java
package arch;

public interface CardInterFace {
    public Card removeFirstElement() throws IllegalStateException;
    public boolean isEmpty();
    public boolean addNewEntry(Object newEntry) throws LockedDeckException;
    public Card[] toArray() throws EmptyDeckException;
    public boolean addNewEntry(Card newEntry) throws LockedDeckException;
    public Card getFirstCard() throws NullPointerException;
    public Card getLastCard() throws NullPointerException;

}
```

## Card.java

```java
package arch;

public class Card extends Node{
    private String  suit, strSuit;
    private boolean isFace;
    private int     cardValue;

    //new Card(1,"h");
    public Card(int pValue, String pSuit) {
        super(pValue);
        cardValue = validateValue(pValue);
        setIsFaceCard();
        validateSuit(pSuit);
    }

    public Card() throws NullPointerException {
        throw new NullPointerException("Empty Card Cannot Be created");
    }

    //card point value restriction
    private int validateValue(int value) throws IllegalStateException{
        if (value >= 1 && value <= 13){
            return value;
        }else{
            throw new IllegalStateException("Invalid CardValue: " + value);
        }
    }

    //nice suit display
    private void validateSuit(String pSuit) throws IllegalStateException{
        String suitLower = pSuit.toLowerCase();
        String tmpsuit = "";
        if (suitLower.equals("s") || suitLower.equals("c") || suitLower.equals("h") || suitLower.
    equals("d")){
            switch (suitLower){
                case "s":
                    tmpsuit = "s"; //<---- changed due to latex error
                    break;
                case "c":
                    tmpsuit = "c";
                    break;
                case "h":
                    tmpsuit = "h";
                    break;
                case "d":
                    tmpsuit = "d";
                    break;
            }
            strSuit = suitLower;
            suit =  tmpsuit;
        }else{
            throw new IllegalStateException("Invalid Suit: " + pSuit);
        }
    }
    public String toString() throws IllegalStateException{                      //representation
        String tmpDenoted = "";
        switch (cardValue) {
            case 1:  tmpDenoted = "A";
                break;
            case 2: case 3: case 4: case 5: case 6: case 7: case 8: case 9:
                tmpDenoted = Integer.toString(cardValue);
                break;
            case 10:  tmpDenoted = "T";
                break;
            case 11:  tmpDenoted = "J";
                break;
            case 12:  tmpDenoted = "Q";
                break;
            case 13:  tmpDenoted = "K";
                break;
            default:
                throw new IllegalStateException("Unknown cardValue: " + cardValue);
        }
        return tmpDenoted+this.suit;
    }
    public int getCardValue() {return cardValue;}                              //get point value e.g. J -> 11
    protected void setCardValue(int v) {cardValue = v;}                        //set a card value  e.g. 11 -> J
    protected void setCardSuit(String s) {                                      //stringify a suit
        strSuit = s;
        validateSuit(s);
    }
    public String getSuit()   {return suit;}                                    //format
    public boolean isFace()   {return isFace;}                                  //is JQK?
    public String getStrSuit(){return strSuit;}                                 //get the suit in "h","s","p" format
    private void setIsFaceCard(){isFace = cardValue >= 11 && cardValue <= 13;}  //set faceCard
}
```

## Deck.java

```java
package arch;
import java.util.Random;
public class Deck<T> extends LinkedList implements CardInterFace {
    private Card next, firstCard, lastCard;
    private final int SHUFFLESIZE = 1000;
    private boolean lock = false;
    private static final String[] SUITS = {"h","d","s","c"};

    //build a standard deck
    public Deck() throws LockedDeckException, NullPointerException{
        for (int r = 1; r <= 13 ; r++) {
            for (int s = 0; s <= 3 ; s++) {
                addNewEntry(new Card(r, SUITS[s]));
            }
        }
    }
    public Deck(boolean shuffle, boolean fill)throws LockedDeckException,
            NullPointerException, EmptyDeckException{
        //optional shuffle and fill
        if (fill) {
            for (int r = 1; r <= 13; r++) {
                for (int s = 0; s <= 3; s++) {
                    addNewEntry(new Card(r, SUITS[s]));
                }
            }
        }
        if (shuffle){
            shuffle();
            lock =true;
        }

    }
    public void shuffle() throws LockedDeckException, EmptyDeckException {
        //deck is locked after shuffle, disabling more shuffle
        if (lock == true)
            throw new LockedDeckException("Deck is locked");
        //empty deck cannot be shuffled
        if (size == 0)
            throw new EmptyDeckException("The Deck is Empty");
        Random random = new Random();
        //shuffle N times
        for (int i = 0; i < size*SHUFFLESIZE; i++) {
            int randomElement = random.nextInt(size - 1);
            int j = 0;
            Card currentCard = firstCard;
            //iterate to the card
            while (j < randomElement && currentCard.getNext() != null) {
                currentCard = (Card) currentCard.getNext();
                j++;
            }
            //insert it to the front
            Card first = (Card) currentCard;
            Card SecondCard = (Card) currentCard.getNext();
            Card thirdCard = (Card) currentCard.getNext().getNext();

            first.setNext(thirdCard);
            SecondCard.setNext(this.firstCard);
            this.firstCard = SecondCard;
            //if we pick the last element set size-2 card to last
            if (randomElement == size - 2){
                this.lastCard = currentCard;
            }
        }
        //lock the deck to avoid shuffle
        lock = true;
    }
    public Card removeFirstElement() throws IllegalStateException{
        if (size == 0){throw new IllegalStateException("Cannot " +
                "remove a Card from an empty deck");}
        //removing the very first card
        if(firstCard != null){
            Card first = (Card) firstCard;
            firstCard =  (Card) firstCard.getNext();
            size--;
            //for integrity
            if (size == 1) updateLastElement();
            //if card removed return with it
            return first;
            //return a BIG null
        }else return null;
    }

    //remove a specific card from the deck
    public void removeFirstElement(Card aCard) throws CardNotFoundException{
        Card currentCard = (Card) firstCard;
        boolean found = false;
        //remove the first element
        if (currentCard.getCardValue() == aCard.getCardValue() &&
                currentCard.getStrSuit() == aCard.getStrSuit()) {
            firstCard = (Card) currentCard.getNext();
            size--;
            found = true;
        }
        while(!found && currentCard.getNext() != null){
            Card nextCard = (Card) currentCard.getNext();

            //if both suit and value are the same //we found the card
            if (nextCard.getCardValue() == aCard.getCardValue() &&
                    nextCard.getStrSuit() == aCard.getStrSuit()) {
                found = true;
                size--;
                //skipp the next card
                currentCard.setNext(nextCard.getNext());
            }else{
                //if no match, get the next card
                currentCard = (Card) currentCard.getNext();
            }
        }//endwhile
        setlastcard();
        if (!found)
            throw new CardNotFoundException(aCard);
    }

    //update the last card safety
    public void setlastcard(){
        Card card = firstCard;
        while(card.getNext() != null) {
            card = (Card) card.getNext();
        }
        lastCard = card;
    }

    //add a new card to the deck
    public boolean addNewEntry(Card newEntry) throws LockedDeckException{
        //when the deck is locked (after shuffle), disable the method
        if (lock){
            throw new LockedDeckException("Deck is Locked, not modifiable");
        }
        Card newCard = newEntry;
        newCard.setNext(firstCard);
        firstCard = newCard;
        size++;
        //if we have only one node, we record it as a last node
        if (getSize() == 1)
            lastCard = newCard;
        return true;
    }

    //array representation of the deck
    public Card[] toArray() throws EmptyDeckException{
        Card[] cardArray =  new Card[size];
        Card[] emptyArray = new Card[0];

        if (isEmpty() == true)
            throw new EmptyDeckException("Deck is empty");

        int counter = 0;
        Card currentCard = firstCard;
        while (currentCard.getNext() != null) {
            cardArray[counter] = currentCard;
            currentCard = (Card) currentCard.getNext();
            counter++;
        }
        return cardArray;
    }
    public int getSize(){                                   //get the size of the deck
        return size;
    }  //return the deck size

    //string representation of the array
    public String toString() throws NullPointerException{
        Card currentCard = firstCard;
        String result = "[";
        //fill the array
        while (currentCard.getNext() != null) {
            result += currentCard.toString()+" ";
            currentCard = (Card) currentCard.getNext();
        }
        if (getLastCard() != null){
            result += currentCard.toString();
        }else{
            throw new NullPointerException("The Deck is Empty");
        }
        //result += "["+currentCard.toString()+"]";
        result += "]";
        return result;
    }

    //grab the first card
    public Card getFirstCard() throws NullPointerException {
        if (firstCard== null)
            throw new NullPointerException("The Deck is Empty");
        return firstCard;
    }

    //grab the last card
    public Card getLastCard() throws NullPointerException  {
        if (firstCard== null)
            throw new NullPointerException("The Deck is Empty");
        return lastCard;
    }
}
```

## LinkedList.java

```java
package arch;

public class LinkedList<T> {
    protected Node<T> firstNode;
    protected Node<T> lastNode;
    protected Integer size;

    public LinkedList() {
        firstNode = null;
        lastNode = null;
        size = 0;
    }

    public Node<T> getFirstNode() {
        return firstNode;
    } //return first node

    public Node<T> getLastNode() {
        return lastNode;
    }//return last node

    public int getSize() {
        return size;
    } //get size

    public boolean isEmpty() {
        return size == 0;
    } //is empty

    protected void updateLastElement() {
        //this method updates the last node, which is required
        // if we remove exactly the last element from the list
        if (getSize() == 0) {
            lastNode = null;
        } else if (getSize() == 1) {
            lastNode = firstNode;
        } else {
            Node<T> currentNode = firstNode;
            while (currentNode != null) {
                if (currentNode.getNext() != null) {
                    currentNode = currentNode.getNext();
                } else {
                    lastNode = currentNode;
                    break;
                }
            }
        }
    }

    //add new element to the data structure
    public boolean addNewEntry(Object newEntry) {
        Node<T> newNode = new Node<T>((T) newEntry);
        newNode.setNext(firstNode);
        firstNode = newNode;
        size++;
        //if we have only one node, we record it as a last node
        if (getSize() == 1)
            lastNode = newNode;
        return true;
    }

    //remove the first element without getting the object
    public T removeFirstElement() throws NullPointerException{
        if(size==0){
            throw new NullPointerException("Empty list");
        }
        if (firstNode != null) {
            T result = firstNode.getData();
            firstNode = firstNode.getNext();
            size--;
            updateLastElement();
            return result;
        } else return null;
    }

    //remove first, return with first
    public boolean removeSpfecific(Object anEntry) {
        Node<T> nodeToRemove = findEntry((T) anEntry);
        if (nodeToRemove.getData() == null) {
            return false;
        }
        Node<T> node = firstNode;

        nodeToRemove.setData(firstNode.getData());
        firstNode = firstNode.getNext();
        size--;
        return true;
    }

    //clear everything
    protected void clear() {
        firstNode.setNext(null);
        lastNode.setNext(null);
        size = 0;
    }

    //find a given entry
    protected Node<T> findEntry(T nodeToFind) {
        Node<T> currentNode = firstNode;
        boolean found = false;

        while (!found && currentNode.getNext() != null) {
            if (currentNode.getData().equals(nodeToFind)) {
                found = true;
                return currentNode;
            } else {
                currentNode = currentNode.getNext();
            }
        }
        if (lastNode.getData() == nodeToFind) {
            return lastNode;
        }
        return null;
    }

    //representation in array
    public T[] toArray() throws EmptyDeckException {
        T[] resultArray = (T[]) new Object[size];

        if (isEmpty() == true)
            return resultArray;

        int counter = 0;
        Node<T> currentNode = firstNode;

        while (currentNode.getNext() != null) {
            resultArray[counter] = currentNode.getData();
            currentNode = currentNode.getNext();
            counter++;
        }
        return resultArray;
    }

    //string representation
    public String toString() {
        if (isEmpty() == true) {
            return "[]";
        }
```

```java
        Node<T> currentNode = firstNode;
        String result = "[";
        while (currentNode.getNext() != null) {
            result += currentNode.getData();
            currentNode = currentNode.getNext();
        }
        if (getLastNode() != null)
            result += lastNode.getData();
        result += "]";
        return result;
    }
}
```

## Node.java

```java
package arch;
//setting up individual nodes, which can point to another element
public class Node<T> {
    private T cardValue;
    private Node<T> next;

    public Node() throws NullPointerException{
        throw new NullPointerException("No data in node!");
    }

    public Node(T dataValue) {
        cardValue = dataValue;
        next = null;
    }

    //get the value of the node
    public T getData() {
        return cardValue;
    }

    //set the node data
    public void setData(T dataValue) {
        cardValue = dataValue;
    }

    //get the next node
    public Node<T> getNext() {
        return next;
    }

    //set the next node
    public void setNext(Node<T> nextNode) {
        next = nextNode;
    }

    //string repr
    public String toString(){
        if (getData() != null) {
            return "Data:\t" + getData()+ "\t->\t" + getNext();
        }else{
            return "Data:\t null";
        }
    }
}
```

## ReplayItem.java

```java
package arch;

public class ReplayItem extends Node{
    private String  data;

    public ReplayItem(String pValue) {
        super(pValue);
        data = pValue;
    }

    public String toString(){
        return data;
    }

    public String getData() {return data;}
}
```

## Replay.java

```java
package arch;

import java.util.Scanner;

public class Replay<T> extends Stack implements StackInterface {
    private ReplayItem firstItem, lastItem;

    //add a new item to replay
    public boolean push(ReplayItem pItem) throws LockedDeckException {
        if (firstItem != null) {
            pItem.setNext(firstItem);
        }
        firstItem = pItem;

        size++;
        //if we have only one node, we record it as a last node
        if (size == 1)
            lastItem = pItem;
        return true;
    }

    //array representation of the replay
    public ReplayItem[] toArray() throws EmptyDeckException {
        ReplayItem[] itemArray = new ReplayItem[size];
        ReplayItem[] emptyArray = new ReplayItem[0];

        if (isEmpty() == true)
            throw new EmptyDeckException("Deck is empty");

        //build the result array
        int counter = 0;
        ReplayItem currentItem = firstItem;
        while (currentItem.getNext() != null) {
            itemArray[counter] = currentItem;
            currentItem = (ReplayItem) currentItem.getNext();
            counter++;
        }
        return itemArray;
    }

    //get the size of the replay
    public int getSize() {                                   //get the size of the deck
        return size;
    }

    //string representation
    public String toString() throws NullPointerException {

        ReplayItem currentItem = firstItem;
        String result = "";
        //result += firstItem.getData()+"\n";
        ReplayItem nextItem = (ReplayItem) currentItem.getNext();

        //fill the array
        while (currentItem.getNext() != null) {
            result = currentItem.getData();
            currentItem = (ReplayItem) currentItem.getNext();
            System.out.println(result);

        }
        System.out.println(lastItem.getData());
        result = lastItem.getData() + result + "\n";

        return "";
    }

    //grab the last replay item
    public ReplayItem getLastItem() throws NullPointerException {
        if (firstItem == null)
            throw new NullPointerException("The Deck is Empty");
        return lastItem;
    }

    //grab the last card and delete it
    public String getLastItemAndDelete(){
        if (size >0){
            ReplayItem item = getLastItem();

        }
        String tmp = "-1";
        ReplayItem currentItem = firstItem;
        for (int i = 0; i < size-1; i++) {
            currentItem = (ReplayItem) currentItem.getNext();
        }
        tmp =  currentItem.getData();
        currentItem.setNext(null);
        size--;
        return tmp;
    }
    //print out in reverse
    public void printHistory(){
        Scanner scan = new Scanner(System.in);
        while(size != 0){
            System.out.println(getLastItemAndDelete());
            System.out.print(">>>");
            scan.nextLine();
        }
        System.out.println(">>>Bye...");
    }
}
```

## StackInterface.java

```java
package arch;

public interface StackInterface {
    public int[] peek();
    public int getSize();
    public boolean push(int[] newEntry);
    public boolean isEmpty();
    public int[] pop ();
    public void clear ();
}
```

## Stack.java

```java
package arch;

public class Stack<T> implements StackInterface{
    protected Node<T> firstNode;
    protected Node<T> lastNode;
    protected int[] data;
    protected int size;

    public Stack() {
        firstNode = null;
        size = 0;
    }

    public int[] peek() {
        return (int[]) firstNode.getData();
    }

    public int getSize() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public boolean push(int[] newEntry) {
        Node<T> newNode = new Node(newEntry);
        newNode.setNext(firstNode);
        int[] arr = newEntry;
        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                int tmp = 0;
                if (arr[i] < arr[j]) {
                    tmp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = tmp;
                }
            }
        }
            newNode.setData((T) arr);
            firstNode = newNode;
            size++;
            if (size == 1) {
                lastNode = newNode;
            }
            return true;
        }

        public int[] pop () {
            if (firstNode != null) {
                int[] result = (int[]) firstNode.getData();
                firstNode = firstNode.getNext();
                size--;
                return result;
            } else return null;
        }

        public void clear () {
            if (firstNode != null) {
                firstNode.setNext(null);
                size = 0;
            }
        }

        public String toString () {
            if (isEmpty()) {
                return "[]";
            }
            String result = "[";
            Node<T> currentNode = firstNode;

            while (currentNode.getNext() != null) {
                for (int i : (int[]) currentNode.getData()) {
                    result += i;
                }
                result += "|";
                currentNode = currentNode.getNext();

            }
            for (int i : (int[]) lastNode.getData()) {
                result += i;
            }
            result += "]";
            return result;
        }
    }
```

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