CardGame.java

```java
import java.util.Collections;
import java.util.LinkedList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * @see CardGame
 *
 *      - CardGame Class is the core class of this project, it's constructor
 *      will take an Integer as number of players and a LinkedList of Integers
 *      which demonstrates the pack, then it would distribute the cards inside
 *      that pack to the players in a round robin fashion and fills the decks
 *      with the remaining cards, it would then pass the created players and
 *      decks lists to their relevant classes, and uses InputOutput class to
 *      write the players initial hand.
 *
 * @Note this class follows the requirements described inside the specification
 *       of this project
 *
 * @author Amirali Famili
 */
public class CardGame {

    final private int timeSlice = 50;
    private int playerNumber;
    private LinkedList<LinkedList<Integer>> decks = new LinkedList<LinkedList<Integer>>();
    private LinkedList<LinkedList<Integer>> players = new LinkedList<LinkedList<Integer>>();
    private LinkedList<Integer> pack;

    /**
     * @see CardGame(int)
     *
     *      - CardGame(int) is the main constructor for CardGame class, it receives
     *      an
     *      integer which indicates the number of players and a LinkedList which
     *      indicates the pack, then it would create players and decks lists with
     *      playerNumber empty LinkedLists inside them, deals the cards to players
     *      and fills the decks with remaining cards from pack, then it would
     *      captures the initial hand with InputOutput class and pass players and
     *      decks to their corresponding classes.
     *
     * @Note if the playerNumber is in the wrong format, constructor will assume a
     *       single player game should be played.
     *
     * @param playerNumber number of players which indicates the size of players and
     *                     decks list
     * @param pack         a LinkedList which should contain all the cards played in
     *                     this game for both players and decks.
     */
    public CardGame(int playerNumber, LinkedList<Integer> pack) {
        if (playerNumber > 0) {
```

```
52          this.playerNumber = playerNumber;
53      } else {
54          this.playerNumber = 1;
55      }
56      this.pack = pack;
57      setPlayers(this.playerNumber);
58      setDecks(this.playerNumber);
59      this.players = dealHands();
60      this.decks = dealDecks();
61      InputOutput output = new InputOutput(players);
62      Player player = new Player(this.players);
63      Card card = new Card(this.decks);
64  }
65
66  /**
67   * @see CardGame()
68   *
69   *      - CardGame() is the default constructor of the Card class, mainly used
70   *      for
71   *      manipulation of instances of this class from the corresponding test
72   *      class.
73   *
74   */
75  public CardGame() {
76      // default constructor
77  }
78
79  /**
80   * @see emptyPack
81   *
82   *      - emptyPack is a void method, used for emptying the pack by assigning it
83   *      to a new LinkedList.
84   */
85  private void emptyPack() {
86      this.pack = new LinkedList<Integer>();
87  }
88
89  /**
90   * @see getPack
91   *
92   *      - getPack is the getter method for pack, used for retrieving the pack.
93   *
94   * @return pack
95   */
96  private LinkedList<Integer> getPack() {
97      return this.pack;
98  }
99
100
101 /**
102  * @see createPack
103  *
104  *      - createPack creates a pack of size 8 times n which consists of integers
105  *      from 1 to n*2 each repeated 4 times .
```

```
106      *      , after the creating of such pack it uses Collections.shuffle to shuffle
107      *      the pack in a random way.
108      *
109      * @Note this is an unimplemented method, it's mostly used within the test
110      *       classes for creating a mock pack instead of receiving a pack file each
111      *       time.
112      *
113      * @param n an integer to determine the size and content of the pack
114      *
115      * @return pack
116      */
117     private LinkedList<Integer> createPack(int n) {
118         if (this.pack == null) {
119             emptyPack();
120         }
121         if (n < 1) {
122             n = 1;
123         }
124         try { // if fails due to the over load of the lists
125             for (int i = 1; i <= (n * 2); i++) {
126                 for (int j = 1; j <= 4; j++)
127                     pack.add(i);
128             }
129         } catch (OutOfMemoryError e) {
130             e.printStackTrace();
131         }
132         Collections.shuffle(pack);
133         return pack;
134     }
135
136     /**
137      * @see get1FromPack
138      *
139      *      - get1FromPack removes and returns the first element of the pack
140      *      LinkedList, it will return -1 if an unexpected error happens
141      *      whilst retrieving the element which is later handled by the program.
142      *
143      * @return the first element of the pack (which also referred to as card in this
144      *         game)
145      */
146     private int get1FromPack() {
147         try {
148             int card = this.pack.poll();
149             return card;
150         } catch (Exception e) {
151             return -1;
152         }
153     }
154
155     /**
156      * @see setPlayers
157      *
158      *      - setPlayers is a void method, it assigns and sets players nested
159      *      LinkedList, each player is represented by the index of their hand inside
```

```
160      *     the
161      *     nested linked list players, this method adds player number times empty
162      *     LinkedLists to the players nested LinkedList.
163      *
164      * @param playerNumber determines the length of the players list to be created
165      */
166     private void setPlayers(int playerNumber) {
167        this.players = new LinkedList<LinkedList<Integer>>();
168        for (int i = 0; i < playerNumber; i++) {
169           this.players.add(new LinkedList<Integer>());
170        }
171     }
172
173     /**
174      * @see setDecks
175      *
176      *     - setDecks is a void method, it assigns and sets decks nested
177      *     LinkedList, each deck is represented by a LinkedList inside the
178      *     nested linked list decks, this method adds player number times empty
179      *     LinkedLists to decks.
180      *
181      * @param playerNumber determines the length of the decks list to be created
182      *
183      * @Note the left deck has the same index as player and right deck is on index
184      *       above that
185      */
186     private void setDecks(int playerNumber) {
187        this.decks = new LinkedList<LinkedList<Integer>>();
188        for (int i = 0; i < playerNumber; i++) {
189           this.decks.add(new LinkedList<Integer>());
190        }
191     }
192
193     /**
194      * @see dealHands
195      *
196      *     - dealHands uses get1FromPack method to deal 1 card at a time to each
197      *     player in the game, until each player's hand has 4 cards.
198      *
199      */
200     private LinkedList<LinkedList<Integer>> dealHands() {
201        for (int i = 0; i < 4; i++) {
202           for (int j = 0; j < playerNumber; j++) {
203              LinkedList<Integer> playerN = this.players.get(j);
204              int card = get1FromPack();
205              if (card != -1) {
206                 playerN.add(card);
207              }
208           }
209        }
210
211        return this.players;
212     }
213
```

```
214     /**
215      * @see dealDecks
216      *
217      *      - dealDecks uses get1FromPack method to deal 1 card at a time to each
218      *      deck in the game, until each deck has 4 cards and the pack is empty.
219      *
220      */
221     private LinkedList<LinkedList<Integer>> dealDecks() {
222         int index = 0;
223         while (!this.pack.isEmpty()) {
224             LinkedList<Integer> deck = this.decks.get(index);
225             int card = get1FromPack();
226             if (card != -1) {
227                 deck.add(card);
228             }
229             index = (index + 1) % playerNumber;
230         }
231         return decks;
232     }
233
234     /**
235      * @see getPlayers
236      *
237      *      - getPlayers is the getter method for players, it retrieves the main
238      *      players list which consists of all player hands in the game.
239      *
240      * @return players, the main list for players hands
241      */
242     private LinkedList<LinkedList<Integer>> getPlayers() {
243         return this.players;
244     }
245
246     /**
247      * @see getDecks
248      *
249      *      - getDecks is the getter method for decks, it retrieves the main decks
250      *      list which consists of all player decks in the game.
251      *
252      * @return decks, the main list for players decks
253      */
254     private LinkedList<LinkedList<Integer>> getDecks() {
255         return this.decks;
256     }
257
258     private static volatile boolean winnerDeclared = false;
259     private volatile Boolean win = false;
260     private final Object lock = new Object();
261     private int counter = 0;
262
263     /**
264      * @see PlayerThread
265      *
266      *      - PlayerThread is a nested class responsible for handling player
267      *      threads, it extends Thread and has a run method in which it keeps
```

```
268      *      executing a player turn until a player wins the game.
269      *
270      * @Note playerThread is an inner class of Player class mainly responsible for
271      *       multi threading of players and handling the threads to avoid any
272      *       problems such as race condition
273      */
274     private class PlayerThread extends Thread {
275
276         /**
277          * @see run
278          *
279          *      - run method for threading the players each player will use this run
280          *      method to play the game from within their own thread.
281          *
282          * @Note this method keeps executing player Turns until a player wins the game.
283          */
284         @Override
285         public void run() {
286
287             if (!players.get(0).isEmpty()) {
288                 while (!win) {
289                     if (counter == -1) {
290                         break;
291                     }
292                     playerTurn();
293                 }
294             } else {
295                 System.out.println("players have not been set please re run the game");
296                 System.exit(0);
297             }
298         }
299
300         /**
301          * @see playerTurn
302          *
303          *      - playerTurn is a void method, it operates the actions made by each
304          *      player (thread) in their turn, including discarding a card to their
305          *      right deck and taking a card from their left deck as well as logging all
306          *      the actions made by them to their relevant files.
307          *
308          * @Note all the actions in this method is thread safe, although all the players
309          *       are attempting to take a turn at the same time.
310          *
311          * @ClassesUsed Card, Player, InputOutput
312          */
313         private synchronized void playerTurn() {
314
315             Card card = new Card(decks);
316             Player player = new Player(players);
317
318             try {
319                 card.getRightDeck((counter + 1) % playerNumber);
320             } catch (IndexOutOfBoundsException e) {
321                 playerTurn();
```

```
322             }
323
324             LinkedList<Integer> hand = player.getPlayer(counter);
325             LinkedList<Integer> leftDeck = card.getLeftDeck(counter);
326             LinkedList<Integer> rightDeck = card.getRightDeck(counter);
327
328             synchronized (lock) {
329                 while (leftDeck.isEmpty()) {
330                     try {
331                         lock.wait(timeSlice);
332                     } catch (InterruptedException e) {
333                     }
334                 }
335
336                 int discard = player.getCard(hand);
337                 if (hand.contains(discard) && !leftDeck.isEmpty() && discard != 0) {
338                     try {
339                         int draw = card.getCardFromLeftDeck(leftDeck);
340
341                         if (draw != -1) {
342                             player.replaceCard(discard, draw, hand);
343                             card.putCardToRightDeck(discard, rightDeck);
344                             InputOutput output = new InputOutput(); // is it better to do this with a constructor ?
345                             output.writeCurrentHand(hand, (counter % playerNumber) + 1);
346                             output.writeDrawsCard(draw, (counter % playerNumber) + 1);
347                             output.writeDiscardsCard(discard, (counter % playerNumber) + 1);
348                             System.out.println("Round :  " + counter + " Player :  " + ((counter % playerNumber) + 1) + " Hand :  "
    + hand);
349                             win = playerWon(hand);
350                         }
351                     } catch (Exception e) {
352                         try {
353                             lock.wait(timeSlice);
354                         } catch (Exception ee) {
355                             Thread.currentThread().interrupt();
356                         }
357                     }
358                 }
359             }
360             counter++;
361         }
362     }
363
364     /**
365      * @see playerWon
366      *
367      *     - playerWon is method that checks the players hands for a win condition
368      *     (if they hold the exact same 4 cards in their hand)
369      *     it does this by creating new instances of player hand and all players
370      *     (since the hand might change with all the threads it needs to capture
371      *     it)
372      *     after that it validates both the hand and players and runs a run only
373      *     once code so that we would have one winner, the winner then notifies
374      *     other players that they have won using InputOutput class and the game
```

```
375     *      will end.
376     *
377     * @Note event with notifyAll method and locks used, threads still seem to be
378     *       running the playerTurn once before the winner is declared, that is the
379     *       reason there are many restrictions in this method and system.exit is
380     *       used since after all players will play once more when the winner is
381     *       declared there is a slight possibility that the game might have two or
382     *       more winners system.exit is used to prevent that from happening.
383     *
384     * @param player represents a player hand which should be checked for winning
385     *               condition
386     *
387     * @ClassesUsed Card, Player, InputOutput
388     *
389     * @return true if player is a winning hand, false if it's not
390     */
391    private synchronized boolean playerWon(LinkedList<Integer> player) {
392        try {
393            if (player.size() != 4) {
394                return false;
395            }
396            LinkedList<Integer> hand = new LinkedList<Integer>(player);
397            if (hand.equals(null) || hand.contains(null)) {
398                return false;
399            }
400            LinkedList<LinkedList<Integer>> allPlayers = new LinkedList<LinkedList<Integer>>(Player.getPlayers());
401
402            if (!winnerDeclared && allPlayers != null) {
403                try {
404                    synchronized (lock) {
405                        if (allPlayers.contains(hand)) {
406                            if (hand.get(0).equals(hand.get(1))
407                                    && hand.get(1).equals(hand.get(2))
408                                    && hand.get(2).equals(hand.get(3))) {
409                                winnerDeclared = true;
410                                int playerIndex = allPlayers.indexOf(hand);
411                                if (playerIndex != -1) {
412                                    System.out.println("Player_" + (playerIndex + 1) + " wins!");
413                                    InputOutput output = new InputOutput(hand, Player.getPlayers(),
414                                            Card.getDecks());
415                                    this.win = true;
416                                    counter = -1;
417                                    System.exit(0);
418                                } else {
419                                    System.out.println("Error: Player not found in finalPlayers");
420                                }
421                            }
422                        }
423                    }
424                } catch (Exception e) {
425                    e.printStackTrace();
426                }
427            }
428
```

```
429         } catch (Exception e) {
430             e.printStackTrace();
431         }
432 
433         return false;
434     }
435 
436     /**
437      * @see startGame
438      *
439      *      - startGame is a void method, it creates a thread pool for as many
440      *      players as the game has ,
441      *      then it would call the PlayerThread Class and assign each player with
442      *      their own thread,
443      *      they execute the run method which has the playerTurn method inside it.
444      *
445      *      - this class is the main class that runs the game.
446      *
447      * @Note I have used ExecutorService for assigning threads I found it a more
448      *       efficient and easier way.
449      *
450      *
451      * @ClassesUsed PlayerThread
452      */
453     private void startGame() {
454         ExecutorService executorService = Executors.newFixedThreadPool(players.size());
455 
456         for (int i = 0; i < players.size(); i++) {
457             PlayerThread playerThread = new PlayerThread();
458             executorService.execute(playerThread);
459         }
460 
461         executorService.shutdown();
462 
463     }
464 
465     /**
466      * @see main
467      *
468      *      - the game can be played from here, the program asks for a player number
469      *      which should be a valid Integer and a location for a pack to load which
470      *      should be a valid pack and an assessable location.
471      *      - It would then pass those to the CardGame class and prepare the
472      *      essential content for running the game, then it would initiate the game
473      *      using the startGame() method.
474      *
475      *
476      * @Note the while loop used is for making sure that player number and number of
477      *       elements in a valid pack match the requirements for running the game.
478      *
479      *
480      * @ClassesUsed InputOutput, CardGame
481      */
482     public static void main(String[] args) {
```

```
483         InputOutput obj = new InputOutput();
484         int playerNumber = obj.getPlayerNumber();
485         LinkedList<Integer> pack = obj.getPackFilePath();
486 
487         // this while loop will make sure that the combination of pack and player number is correct
488         // Note that the game is runnable with any integer values but it just might never end !
489         while (playerNumber * 8 >= pack.size()) {
490             System.out.println("Your pack should have at least " + playerNumber * 8
491                     + " cards inside it otherwise the game could not run");
492             System.out.println(
493                     "Please either decrease the number of players or change the pack file to match the requirements of the
    game");
494 
495             playerNumber = obj.getPlayerNumber();
496             pack = obj.getPackFilePath();
497         }
498 
499 
500         CardGame cardGame = new CardGame(playerNumber, pack);
501         cardGame.startGame(); // game starts
502     }
503 }
504 
```

Card.java

```
1
2  import java.util.LinkedList;
3
4  /**
5   * @see Card
6   *
7   *      - Class Card is used for managing decks, it receives the initial decks
8   *      from CardGame class and communicates to the main game method with it,
9   *      it's also responsible for deck related operations such as taking an
10  *      element from the left deck and inserting an element to the right deck.
11  *
12  * @Note decks live information is only available within this class and can be
13  *       easily accessed using a static reference
14  *
15  * @author Amirali Famili
16  */
17 public class Card {
18
19     private static LinkedList<LinkedList<Integer>> decks = new LinkedList<LinkedList<Integer>>();
20     protected int deckNumber;
21
22     /**
23      * @see Card
24      *
25      *      - Card(LinkedList<LinkedList<Integer>>), is the main constructor for
26      *      the Card
27      *      Class, it takes a nested LinkedList representing decks which
28      *      should be their initial deck.
29      *
30      * @param decks a nested LinkedList consisting of all the decks of players
31      *              within the game
32      */
33     protected Card(LinkedList<LinkedList<Integer>> decks) {
34         this.decks = decks;
35         this.deckNumber = decks.size();
36     }
37
38     public Card() {
39         // default constructor
40     }
41
42     /**
43      * @see getDecks
44      *
45      *      - getDecks(LinkedList<LinkedList<Integer>>) is a synchronized method, it
46      *      returns the current decks of players.
47      *
48      * @return the main decks LinkedList which contains all the player decks
49      */
50     protected static synchronized LinkedList<LinkedList<Integer>> getDecks() {
51         return decks;
```

```
52      }
53
54      /**
55       * @see setDecks
56       *
57       *      - setDecks is an unimplemented synchronized void method, it receives an integer n
58       *      which represents the number of players and creates a decks LinkedList
59       *      with n
60       *      nested LinkedList inside it.
61       *
62       */
63      private synchronized static void setDecks(int n) {
64          decks = new LinkedList<>();
65          if (n <= 0) {
66              n = 1;
67          }
68          for (int i = 0; i < n; i++) {
69              decks.add(new LinkedList<Integer>());
70          }
71      }
72
73      /**
74       * @see getLeftDeck
75       *
76       *      - getLeftDeck(int) is a synchronized method, it receives an index and
77       *      calculates the players left deck and returns it.
78       *
79       * @Note player and their left deck have the same index
80       *
81       * @param index a positive integer which should always return a deck from decks
82       *
83       * @return the leftDeck of the player whom the index is associated with
84       */
85      protected synchronized LinkedList<Integer> getLeftDeck(int index) {
86          if (index < 0) {
87              index = 0;
88          }
89          return decks.get((index % deckNumber));
90      }
91
92      /**
93       * @see getRightDeck
94       *
95       *      - getRightDeck(int) is a synchronized method, it receives an index and
96       *      returns the rightDeck of the player whom the index is associated with.
97       *
98       * @Note player right deck is one index above player's index.
99       *
100      * @param index a positive integer which should always return a deck from decks
101      *
102      * @return the rightDeck of the player whom the index is associated with
103      */
104     protected synchronized LinkedList<Integer> getRightDeck(int index) {
105         if (index < 0) {
```

```
106             index = 0;
107         }
108         return decks.get((((index + 1) % deckNumber)));
109     }
110 
111     /**
112      * @see getCardFromLeftDeck
113      *
114      *      - getCardFromLeftDeck(LinkedList<Integer>) is a synchronized method, it
115      *      receives a LinkedList (leftDeck) and removes and returns the first card
116      *      (integer) of the LinkedList.
117      *
118      * @param leftDeck the left deck of a player which has the same index as theirs
119      *
120      * @return the rightDeck of the player whom the index is associated with
121      */
122     protected synchronized int getCardFromLeftDeck(LinkedList<Integer> leftDeck) {
123 
124         try {
125             return leftDeck.poll();
126         } catch (Exception e) {
127             return -1;
128         }
129     }
130 
131     /**
132      * @see putCardToRightDeck
133      *
134      *      - putCardToRightDeck(LinkedList<Integer>) is a synchronized void method,
135      *      it
136      *      receives a LinkedList (rightDeck) and a Integer (the card that player
137      *      discarded), and it simply adds the discarded card to the rightDeck.
138      *
139      * @param rightDeck the right deck of a player
140      *
141      * @return the rightDeck of the player whom the index is associated with
142      */
143     protected synchronized void putCardToRightDeck(int discard, LinkedList<Integer> rightDeck) {
144         if (rightDeck != null) {
145             rightDeck.add(discard);
146         }
147     }
148 }
149 
```

Player.java

```
1
2  import java.util.*;
3
4  /**
5   * @see Player
6   *
7   *      - Player Class is one of the core classes of this project, it receives a
8   *      LinkedList representing players from it's constructor and interacts with
9   *      the game using players list and operates player related functions to
10  *      keep the players list updated.
11  *
12  * @Note the length of the nested LinkedList players represents the number of
13  *       players currently playing the game..
14  *
15  * @author Amirali Famili
16  */
17 public class Player {
18    protected int playerNumber;
19    private static LinkedList<LinkedList<Integer>> players = new LinkedList<LinkedList<Integer>>();
20
21    /**
22     * @see Player
23     *
24     *      - Player(LinkedList<LinkedList<Integer>>), is the main constructor for
25     *      the Player
26     *      Class, it takes a nested LinkedList representing player hands which
27     *      should be their initial hand.
28     *
29     * @param players a nested LinkedList consisting of all the hands of players
30     *                within the game
31     */
32    protected Player(LinkedList<LinkedList<Integer>> players) {
33       this.playerNumber = players.size();
34       this.players = players;
35    }
36
37    /**
38     * @see replaceCard
39     *
40     *      - replaceCard is a void method, it removes the card chosen by the player
41     *      to be discarded and inserts the card obtained from their leftDeck to
42     *      replace it.
43     *
44     * @Note this synchronized method will make sure that the operation of adding
45     *       and discarding from hands is atomic.
46     *
47     * @param discard is the card that player has chosen to discard
48     * @param draw    the card they have drawn from their leftDeck
49     * @param hand    represents a player's hand which the operations should be
50     *                executed for
51     *
```

```
52      */
53     protected synchronized void replaceCard(int discard, int draw, LinkedList<Integer> hand) {
54        if (hand != null && discard >= 0 && draw >= 0) {
55           if (hand.contains(discard)) {
56              hand.remove(hand.indexOf(discard));
57              hand.add(draw);
58           }
59        }
60     }
61
62     public Player() {
63        // default constructor
64     }
65
66     /**
67      * @see getPlayer
68      *
69      *      - getPlayer will return the current players hand with a given index, the
70      *      index represents the round number in cardGame and mod playerNumber we
71      *      get the player's index.
72      *
73      *
74      * @param index a Positive Integer which should always return a player hand.
75      *
76      * @return the player whom the index is associated with
77      *
78      */
79     protected synchronized LinkedList<Integer> getPlayer(int index) {
80        return this.players.get((index % playerNumber));
81     }
82
83     /**
84      * @see hasDuplicates
85      *
86      *      - hasDuplicates(LinkedList) is a synchronized method for checking if a
87      *      LinkedList of integers had duplicated values within it, .
88      *      it returns true if it finds a duplicate and false if there is no element
89      *      being repeated inside the LinkedList.
90      *
91      * @param hand a LinkedList consisting integers
92      *
93      * @return true or false
94      */
95     protected synchronized boolean hasDuplicates(LinkedList<Integer> hand) {
96        Set<Integer> dup = new HashSet<>();
97
98        try {
99           for (Integer card : hand) {
100             if (!dup.add(card)) {
101                return true;
102             }
103          }
104       } catch (Exception e) {
105          return false;
```

```
106         }
107         return false;
108     }
109 
110     /**
111      * @see getCard
112      *
113      *      - getCard(LinkedList) is a synchronized method, it checks the LinkedList
114      *      passed to it as an argument and it returns
115      *      the first occurrence of an element that has not been repeated within
116      *      hand with the help of hasDuplicate method.
117      *      - this method returns the last element of the hand if it fails to find a
118      *      duplicate,
119      *      if there is a null value it returns 0 which reports a serious problem
120      *      within the system.
121      *
122      *
123      * @param hand a LinkedList consisting integers
124      *
125      * @return an integer representing the card to be removed by the player
126      */
127     protected synchronized int getCard(LinkedList<Integer> hand) {
128         try {
129             if (hand.contains(null) || hand.isEmpty()) {
130                 return 0;
131             }
132 
133         } catch (Exception e) {
134             return 0;
135         }
136         try {
137             if (hasDuplicates(hand)) {
138                 Map<Integer, Integer> cardCount = new HashMap<>();
139                 for (int card : hand) {
140                     cardCount.put(card, cardCount.getOrDefault(card, 0) + 1);
141                 }
142 
143                 for (int card : hand) {
144                     if (cardCount.get(card) == 1) {
145                         return card;
146                     }
147                 }
148             }
149         } catch (Exception e) {
150             return 0;
151         }
152 
153         return hand.getLast(); // it's better if it was in random
154     }
155 
156     /**
157      * @see getPlayers
158      *
159      *      - getPlayers is a synchronized method, it returns the current players
```

```
160         *      list which contains all the hands.
161         *
162         * @return the current players of the game (main LinkedList)
163         */
164        protected synchronized static LinkedList<LinkedList<Integer>> getPlayers() {
165            return players;
166        }
167
168        /**
169         * @see setPlayers
170         *
171         *      - setPlayers is an unimplemented synchronized void method, it receives an integer n
172         *      which represents the number of players and creates a players LinkedList
173         *      with n
174         *      nested LinkedList inside it.
175         *
176         */
177        protected synchronized static void setPlayers(int n) {
178            players = new LinkedList<>();
179            if (n <= 0) {
180                n = 1;
181            }
182            for (int i = 0; i < n; i++) {
183                players.add(new LinkedList<Integer>());
184            }
185        }
186
187    }
```

InputOutput.java

```java
import java.util.Scanner;
import java.util.LinkedList;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.util.List;

/**
 * @see InputOutput
 *
 *      - InputOutput Class is responsible for every interaction the program has
 *      with terminal, and text files of players and decks, it gets the pack
 *      file and validates it and write it into a LinkedList, it also receives
 *      the number of players and validates it.
 *      This class will create player and deck files, delete the old ones and
 *      write to the new ones with the correct format mentioned in the
 *      specification.
 *
 * @Note since creating, deleting and writing to files with threads could lead
 *       to serious compacts almost all methods inside this class are
 *       synchronized and the class could be counted as a thread safe class.
 *
 * @author Amirali Famili
 */
public class InputOutput {

    /**
     * @see getPlayerNumber
     *
     *      - getPlayerNumber is synchronized method, it asks for the number of
     *      players from the terminal and validates the number, if the input entered
     *      by the user is invalid it would recursively call it self until it
     *      receives the correct input.
     *
     * @return the number of players the game should have
     */
    protected synchronized int getPlayerNumber() {
        Scanner scan = new Scanner(System.in);

        System.out.print("\nPlease enter the number of players : ");
        String playerNum = scan.nextLine();

        try {
            int num = Integer.parseInt(playerNum);
            if (num < 1) {
                System.out.println("Please enter a positive Integer as player number: \n");
                return getPlayerNumber();
            } else {
```

```
52              return num;
53          }
54      } catch (NumberFormatException e) {
55          System.out.println("Please enter a valid number as number of players : \n");
56          return getPlayerNumber();
57      }
58  }
59
60  /**
61   * @see getPackFilePath
62   *
63   *      - getPackFilePath is synchronized method, it asks for the location of
64   *      the pack to load, if the path entered is invalid it would recursively
65   *      ask the user to enter a valid path until it finds the file.
66   *
67   * @Note this method is getting help form readInputPack method to put the file
68   *       which contains pack into a LinkedList, the method will readInputPack
69   *       will check the validity of the pack and if the pack is not in the right
70   *       format it would recursively call this method again to get another valid
71   *       pack file.
72   *
73   * @return the pack which contains all the cards the game should be played with.
74   */
75  protected synchronized LinkedList<Integer> getPackFilePath() {
76      LinkedList<Integer> pack = new LinkedList<Integer>();
77      System.out.print("\nPlease enter location of pack to load: ");
78      Scanner scanner = new Scanner(System.in);
79      String filePath = scanner.nextLine();
80      Path path = Paths.get(filePath);
81
82      if (Files.exists(path)) {
83          pack = readInputPack(filePath);
84      } else {
85          System.out.println("Entered location is Not Valid !");
86          getPackFilePath();
87      }
88      return pack;
89  }
90
91  /**
92   * @see readInputPack
93   *
94   *      - readInputPack is synchronized method, it receives a valid filePath and
95   *      validates the content of that file to make sure that, the content of the
96   *      file patch the requirements mentioned in the specification, then it
97   *      would return the LinkedList pack created by the content of the file.
98   *
99   * @Note if something goes wrong whilst adding the content of the file to the
100  *       LinkedList, the code will assume there is something wrong with the pack
101  *       and asks for another valid pack using getPackFilePath method.
102  *
103  * @param filePath a valid file path which points to the location of the pack
104  *
105  * @return the pack which contains all the cards the game should be played with.
```

```
106      */
107     private synchronized LinkedList<Integer> readInputPack(String filePath) {
108         LinkedList<Integer> pack = new LinkedList<Integer>();
109 
110         try {
111             Path path = Paths.get(filePath);
112             List<String> lines = Files.readAllLines(path);
113 
114             for (String line : lines) {
115                 if (line.trim().isEmpty()) {
116                     continue;
117                 }
118                 int value = Integer.parseInt(line.trim());
119                 if (value < 0) {
120                     continue;
121                 }
122                 pack.add(value);
123             }
124         } catch (Exception e) {
125             System.out.println("Pack file does not have the right format please enter another file name");
126             getPackFilePath();
127         }
128         return pack;
129     }
130 
131     private int playerNumber;
132     private LinkedList<LinkedList<Integer>> players = new LinkedList<LinkedList<Integer>>();
133     private final Object lock = new Object();
134 
135     /**
136      * @see InputOutput
137      *
138      *      - InputOutput (LinkedList<LinkedList<Integer>>) is the initial
139      *      constructor of the InputOutput class, it's called inside the CardGame
140      *      constructor at the start of the game, it assigns the players as well as
141      *      player number, it also deletes the deck and player files and create new
142      *      ones for as many player number as the game has.
143      *      Then it would set the players initial hand which is the players
144      *      LinkedList that has been passed down to it.
145      *
146      * @Note this constructor is used for making an automated approach for setting
147      *       players initial hand.
148      *
149      * @param players the main players list at the start of the game which contains
150      *                players initial hands and it's length represents the number of
151      *                players.
152      */
153     public InputOutput(LinkedList<LinkedList<Integer>> players) {
154         synchronized (lock) {
155             this.players = players;
156             this.playerNumber = players.size();
157             deletePlayerFiles();
158             deleteDeckFiles();
159             createPlayerFiles();
```

```
160             createDeckFiles();
161             initialHand(this.players);
162         }
163     }
164
165     public InputOutput() {
166         // default constructor
167     }
168
169     /**
170      * @see InputOutput
171      *
172      *      - InputOutput ((LinkedList<Integer> , LinkedList<LinkedList<Integer>> ,
173      *      LinkedList<LinkedList<Integer>> ) is a constructor of InputOutput class
174      *      used at the end of the game, when the game is won their constructor is
175      *      called and it informs other players that a player has won and it writes
176      *      the decks final content into their corresponding files.
177      *
178      * @param players the final state of the players list which contains all the
179      *                player hands
180      * @param decks   the final state of the decks list which contains all the
181      *                player decks
182      * @param winner  this list represents the hand of the player who has won the
183      *                game.
184      */
185     public InputOutput(LinkedList<Integer> winner, LinkedList<LinkedList<Integer>> players,
186             LinkedList<LinkedList<Integer>> decks) {
187         writeDeckContents(decks);
188         writeEndGame(players, winner);
189     }
190
191     /**
192      * @see writeEndGame
193      *
194      *      - writeEndGame(LinkedList<LinkedList<Integer>> , LinkedList<Integer> )
195      *      is a synchronized void method, it takes the players list and the winner
196      *      hand and write the necessary information for informing other players and
197      *      their state in the game in players files.
198      *
199      * @param players the final state of the players list which contains all the
200      *                player hands
201      * @param winner  this list represents the hand of the player who has won the
202      *                game.
203      */
204     private synchronized void writeEndGame(LinkedList<LinkedList<Integer>> players, LinkedList<Integer> winner)
    {
205         if (players == null) {
206             players = new LinkedList<>();
207         }
208         if (players.contains(null)) {
209             players.remove(null);
210         }
211         if (winner == null) {
212             winner = new LinkedList<>();
```

```
213         }
214         LinkedList<LinkedList<Integer>> allPlayers = new LinkedList<LinkedList<Integer>>(players);
215         LinkedList<Integer> Winner = new LinkedList<Integer>(winner);
216         int index = 1;
217         for (LinkedList<Integer> player : allPlayers) {
218           try {
219             File newFile = new File("players/player" + index + "_output.txt");
220             if (newFile.exists()) {
221               try (FileWriter writer = new FileWriter(newFile, true)) {
222                 if (player != Winner) {
223                   writer.write("player " + (allPlayers.indexOf(Winner) + 1) + " has informed player "
224                       + index + " that player " + (allPlayers.indexOf(Winner) + 1)
225                       + " has won" + "\n");
226                   writer.write("player " + index + " exits\n");
227                   writer.write("player " + index + " final hand: " + cardsToString(player) + "\n");
228                 } else {
229                   writer.write("player " + index + " wins\n");
230                   writer.write("player " + index + " exits\n");
231                   writer.write("player " + index + " final hand: " + cardsToString(player) + "\n");
232                 }
233
234               } catch (IOException e) {
235                 System.out.println(
236                     "Error occurred whilst writing the hand " + cardsToString(player) + " for player "
237                         + player);
238               }
239             } else {
240               System.out.println(
241                   "File " + "players/player" + (allPlayers.indexOf(player) + 1)
242                       + "_output.txt does not exists");
243             }
244           } catch (Exception e) {
245             System.out.println("Error occurred whilst writing the winner information to file players/player"
246                 + (allPlayers.indexOf(player) + 1)
247                 + "_output.txt");
248           }
249           index++;
250         }
251       }
252
253       /**
254        * @see writeDrawsCard
255        *
256        *     - writeDrawsCard(int, int)
257        *     is a synchronized void method, receives an integer representing the card
258        *     that has been drawn by the player and an integer which represents the
259        *     player's index + 1, then it simply write a report to the text file
260        *     informing the user that which player has draw what card from which deck.
261        *
262        * @Note player's index + 1 is used directly to avoid calling the entire players
263        *      list, it represents, player, player's file and their left deck.
264        * @Note player and leftDeck both have the same index.
265        *
266        * @param drawnCard is an integer which represents the card that a certain
```

```
267     *                player has drawn form their leftDeck
268     * @param player    is an integer which represents the player's number, it's
269     *                file name and it's left deck
270     */
271    protected synchronized void writeDrawsCard(int drawnCard, int player) {
272       try {
273          File newFile = new File("players/player" + player + "_output.txt");
274          if (newFile.exists()) {
275             try (FileWriter writer = new FileWriter(newFile, true)) {
276                writer.write("player " + player + " draws a " + drawnCard + " from deck " + player + "\n");
277             } catch (IOException e) {
278                System.out.println(
279                      "Error occurred whilst writing the drawn card " + drawnCard + " for player " + player);
280             }
281          } else {
282             System.out.println("File " + "players/player" + player + "_output.txt does not exists");
283          }
284       } catch (Exception e) {
285          System.out.println("Error occurred whilst writing the player's drawn card to file players/player" + player
286                + "_output.txt");
287          e.printStackTrace();
288       }
289    }
290
291    /**
292     * @see writeDiscardsCard
293     *
294     *     - writeDiscardsCard(int, int)
295     *     is a synchronized void method, receives an integer representing the card
296     *     that has been discarded by the player and an integer which represents
297     *     the
298     *     player's index + 1, then it simply write a report to the text file
299     *     informing the user that which player has discarded what card from which
300     *     deck.
301     *
302     * @Note player's index + 1 is used directly to avoid calling the entire players
303     *      list, it represents, player, player's file and their left deck.
304     * @Note player's right deck's index is player index + 1.
305     *
306     * @param discardedCard is an integer which represents the card that a certain
307     *                player has discarded form their hand
308     * @param player       is an integer which represents the player's number, it's
309     *                file name and it's left deck as well as right deck (is
310     *                index + 1)
311     */
312    protected synchronized void writeDiscardsCard(int discardedCard, int player) {
313       try {
314          File newFile = new File("players/player" + player + "_output.txt");
315          if (newFile.exists()) {
316             try (FileWriter writer = new FileWriter(newFile, true)) {
317                writer.write(
318                      "player " + player + " discards a " + discardedCard + " from deck " + (player + 1) + "\n");
319             } catch (IOException e) {
320                System.out.println(
```

```
                    "Error occurred whilst writing the discarded card " + discardedCard + " for player "
                            + player);
                }
            } else {
                System.out.println("File " + "players/player" + player + "_output.txt does not exists");
            }
        } catch (Exception e) {
            System.out.println("Error occurred whilst writing the player's discarded card to file players/player"
                    + player + "_output.txt");
            e.printStackTrace();
        }
    }

    /**
     * @see writeCurrentHand
     *
     *      - writeCurrentHand(LinkedList<Integer>, int)
     *      is a synchronized void method, it receives a LinkedList which is the
     *      player's hand and an integer which represents the player's number (index
     *      + 1 in players), then it would write the player's current hand inside
     *      the correct file which points to the player.
     *
     * @Note player's index + 1 is used directly to avoid calling the entire players
     *       list, it represents, player, player's file and their left deck.
     *
     * @param hand   is a LinkedList representing player's current hand in the game
     * @param player is an integer which represents the player's number, it's
     *               file name and it's left deck as well as right deck (is
     *               index + 1)
     */
    protected synchronized void writeCurrentHand(LinkedList<Integer> hand, int player) {
        if (hand != null) {
            LinkedList<Integer> staticHand = new LinkedList<>(hand);
            try {
                File newFile = new File("players/player" + player + "_output.txt");
                if (newFile.exists()) {
                    try (FileWriter writer = new FileWriter(newFile, true)) {
                        writer.write("player " + player + " current hand is " + cardsToString(staticHand) + "\n");
                    } catch (IOException e) {
                        System.out.println(
                                "Error occurred whilst writing the hand " + cardsToString(staticHand) + " for player "
                                        + player);
                    }
                } else {
                    System.out.println("File " + "players/player" + player + "_output.txt does not exist");
                }
            } catch (Exception e) {
                System.out.println(
                        "Error occurred whilst writing the player's current hand to file players/player" + player
                                + "_output.txt");
                e.printStackTrace();
            }
        }
    }
```

```
375
376     /**
377      * @see writeDeckContents
378      *
379      *      - writeDeckContents(LinkedList<LinkedList<Integer>>)
380      *      is a synchronized void method, it receives a nested LinkedList which
381      *      represents all the decks which exists in the game, then it would simply
382      *      write the content of each nested list (deck) inside their own files.
383      *
384      * @Note this method will only produce a single line of text inside each deck
385      *       file which indicates the content of each list.
386      *
387      * @param decks is a nested LinkedList which contains all the decks used inside
388      *              the game at their final state.
389      */
390     private synchronized void writeDeckContents(LinkedList<LinkedList<Integer>> decks) {
391         int index = 1;
392         if (decks != null && !decks.isEmpty()) {
393             LinkedList<LinkedList<Integer>> finalDecks = new LinkedList<LinkedList<Integer>>(decks);
394             for (LinkedList<Integer> deck : finalDecks) {
395                 try {
396                     File newFile = new File("decks/deck" + index + "_output.txt");
397                     if (newFile.exists()) {
398                         try (FileWriter writer = new FileWriter(newFile, true)) {
399                             if (!deck.isEmpty()) {
400                                 writer.write(
401                                         "deck " + index + " contents " + cardsToString(deck)
402                                                 + "\n");
403                             } else {
404                                 writer.write("deck " + index + " is Empty " + "\n");
405                             }
406
407                         } catch (IOException e) {
408                             System.out.println(
409                                     "Error occurred whilst writing the final deck " + cardsToString(deck) + " for deck "
410                                             + (finalDecks.indexOf(deck) + 1));
411                         }
412                     } else {
413                         System.out.println(
414                                 "File " + "decks/deck" + (finalDecks.indexOf(deck) + 1)
415                                         + "_output.txt does not exists");
416                     }
417                 } catch (Exception e) {
418                     System.out.println(
419                             "Error occurred whilst writing the final deck to file decks/deck" + index
420                                     + "_output.txt");
421                     e.printStackTrace();
422                 }
423                 index++;
424             }
425         }
426     }
427
428     /**
```

```
429     * @see createPlayerFiles
430     *
431     *      - createPlayerFiles()
432     *      is a synchronized void method, it creates player files for as many
433     *      players as we have in each game, and it would leave them empty to be
434     *      modified later.
435     *
436     */
437    private synchronized void createPlayerFiles() {
438        if (playerNumber < 1) {
439            playerNumber = 1;
440        }
441        for (int i = 1; i <= playerNumber; i++) {
442            try {
443                File newFile = new File("players/player" + i + "_output.txt");
444                if (newFile.exists()) {
445                    try (FileWriter writer = new FileWriter(newFile)) {
446                        writer.write("");
447                    } catch (IOException e) {
448                        System.out.println("Error occurred while trying to clear file for player " + i);
449                    }
450                } else {
451                    newFile.createNewFile();
452                    System.out.println("File created: " + newFile.getName());
453                }
454            } catch (IOException e) {
455                System.out.println("Error occurred while trying to create file for player " + i);
456            }
457        }
458    }
459
460    /**
461     * @see createDeckFiles
462     *
463     *      - createDeckFiles()
464     *      is a synchronized void method, it creates deck files for as many
465     *      decks (or players) as we have in each game, and it would leave them
466     *      empty to be
467     *      modified later.
468     *
469     */
470    private synchronized void createDeckFiles() {
471        if (playerNumber < 1) {
472            playerNumber = 1;
473        }
474        for (int i = 1; i <= playerNumber; i++) {
475            try {
476                File newFile = new File("decks/deck" + i + "_output.txt");
477                if (newFile.exists()) {
478                    try (FileWriter writer = new FileWriter(newFile)) {
479                        writer.write("");
480                    } catch (IOException e) {
481                        System.out.println("Error occurred while trying to clear file for deck " + i);
482                    }
```

```
483             } else {
484                 newFile.createNewFile();
485                 System.out.println("File created: " + newFile.getName());
486             }
487         } catch (IOException e) {
488             System.out.println("Error occurred while trying to create file for deck " + i);
489         }
490     }
491 }
492 
493 /**
494  * @see deleteDeckFiles
495  *
496  *      - deleteDeckFiles()
497  *      is a synchronized void method, it deletes all the deck files from
498  *      previous run of the game.
499  *
500  */
501 private synchronized void deleteDeckFiles() {
502     int counter = 1;
503     do {
504         File newFile = new File("decks/deck" + counter + "_output.txt");
505         if (newFile.exists()) {
506             try {
507                 Files.delete(Paths.get("decks/deck" + counter + "_output.txt"));
508             } catch (IOException e) {
509                 System.out
510                         .println("Error occurred while trying to delete file decks/deck" + counter + "_output.txt");
511             }
512             counter++;
513         } else {
514             break;
515         }
516     } while (true);
517 }
518 
519 /**
520  * @see deletePlayerFiles
521  *
522  *      - deletePlayerFiles()
523  *      is a synchronized void method, it deletes all the player files from
524  *      previous run of the game.
525  *
526  */
527 private synchronized void deletePlayerFiles() {
528     int counter = 1;
529     do {
530         File newFile = new File("players/player" + counter + "_output.txt");
531         if (newFile.exists()) {
532             try {
533                 Files.delete(Paths.get("players/player" + counter + "_output.txt"));
534             } catch (IOException e) {
535                 System.out
536                         .println("Error occurred while trying to delete file players/player" + counter
```

```
537                             + "_output.txt");
538                 }
539                 counter++;
540             } else {
541                 break;
542             }
543         } while (true);
544     }
545 
546     /**
547      * @see handToString
548      *
549      *      - handToString(LinkedList) is used for
550      *      converting LinkedList element to a String,
551      *      and then returning the String representation of the values.
552      *
553      * @param hand a LinkedList consisting integers
554      *
555      * @return the string representation of the values within the hand LinkedList
556      */
557     private synchronized String cardsToString(LinkedList<Integer> hand) {
558         StringBuilder sHand = new StringBuilder();
559         if (hand == null || hand.isEmpty()) {
560             return "";
561         }
562         LinkedList<Integer> handCopy = new LinkedList<>(hand);
563         for (Integer card : handCopy) {
564             if (card != null) {
565                 sHand.append(card.intValue()).append(" ");
566             } else {
567                 sHand.append(" ");
568             }
569         }
570         return sHand.toString();
571     }
572 
573     /**
574      * @see initialHand
575      *
576      *      - initialHand(LinkedList<LinkedList<Integer>>) is a synchronized void
577      *      method, it receives the main players list at the start of the game and
578      *      it would set the players initial hand for each player in the game in
579      *      their corresponding files.
580      *
581      */
582     private synchronized void initialHand(LinkedList<LinkedList<Integer>> players) {
583         int index = 1;
584         if (players != null && !players.isEmpty()) {
585             for (LinkedList<Integer> player : players) {
586                 try (FileWriter writer = new FileWriter("players/player" + index + "_output.txt",
587                         true)) {
588                     if (player != null) {
589                         writer.write(
```

```
                        "player " + index + " initial hand " + cardsToString(player) + "\n");
                }
            } catch (IOException e) {
                System.out.println("Error occurred while writing player's initial hand : " + player + " to file : "
                        + "players/player" + index + "_output.txt");
            }
            index++;
        }
    }
}
```

testCardGame.java

```
1
2
3  import java.lang.reflect.Field;
4  import java.lang.reflect.Method;
5  import java.util.LinkedList;
6  import org.junit.BeforeClass;
7  import org.junit.Test;
8  import org.junit.After;
9  import org.junit.AfterClass;
10
11 import static org.junit.Assert.*;
12
13 /**
14  * @see testCardGame
15  *
16  *      - Class testCardGame is used for testing all the methods and objects as
17  *      well
18  *      as the constructors of the
19  *      CardGame Class, It checks the code for certain exceptions that could
20  *      occur.
21  *      It's also testing the main game method.
22  *
23  * @Note most test methods within this test class are testing multiple aspects
24  *       of the class such as other methods and objects
25  *
26  * @Note For all the methods inside this test class, java reflection is used to
27  *       access CardGame class's private instances.
28  *
29  * @author Amirali Famili
30  */
31 public class testCardGame {
32
33     /**
34      * @see testEmptyPack
35      *
36      *      - testEmptyPack is a void method, it creates a pack and then checks the
37      *      size of the created pack
38      *      then, it would empty the pack and check that it actually emptied inside
39      *      CardGame Class.
40      *
41      * @link CardGame.java
42      *
43      * @CardGameClassInstance cardGame
44      * @CardGameClassMethods createPack(int), emptyPack(), getPack()
45      */
46     @Test
47     public void testEmptyPack() {
48
49         try {
50             CardGame cardGame = new CardGame();
51             Class<?> cardGameClass = cardGame.getClass();
```

```
52            Method emptyPackMethod = cardGameClass.getDeclaredMethod("emptyPack");
53            Method createPackMethod = cardGameClass.getDeclaredMethod("createPack", int.class);
54            Method getPackMethod = cardGameClass.getDeclaredMethod("getPack");
55
56            createPackMethod.setAccessible(true);
57            getPackMethod.setAccessible(true);
58            emptyPackMethod.setAccessible(true);
59
60            createPackMethod.invoke(cardGame, 5);
61            LinkedList<Integer> pack = (LinkedList<Integer>) getPackMethod.invoke(cardGame);
62            assertEquals(5 * 8, pack.size());
63
64            emptyPackMethod.invoke(cardGame);
65
66            assertTrue(((LinkedList<Integer>) getPackMethod.invoke(cardGame)).isEmpty());
67        } catch (Exception e) {
68            fail("testEmptyPack Failed");
69        }
70    }
71
72    /**
73     * @see testGetPack
74     *
75     *      - testGetPack is a void method, it creates a pack and checks for it's
76     *      size, then it would remove the
77     *      first element of the pack and check with getPack method that it's value
78     *      is updated within CardGame Class.
79     *
80     * @link CardGame.java
81     *
82     * @CardGameClassInstance cardGame
83     * @CardGameClassMethods createPack(int), getPack()
84     */
85    @Test
86    public void testGetPack() {
87
88        try {
89            CardGame cardGame = new CardGame();
90            Class<?> cardGameClass = cardGame.getClass();
91            Method createPackMethod = cardGameClass.getDeclaredMethod("createPack", int.class);
92            Method getPackMethod = cardGameClass.getDeclaredMethod("getPack");
93
94            createPackMethod.setAccessible(true);
95            getPackMethod.setAccessible(true);
96
97            createPackMethod.invoke(cardGame, 34);
98            LinkedList<Integer> pack = (LinkedList<Integer>) getPackMethod.invoke(cardGame);
99
100           assertTrue(pack.size() == 34 * 8);
101           pack.removeFirst();
102
103           assertTrue(((LinkedList<Integer>) getPackMethod.invoke(cardGame)).size() == ((34 * 8) - 1));
104       } catch (Exception e) {
105           fail("testGetPack Failed");
```

```
106         }
107     }
108 
109     /**
110      * @see testCreatePack
111      *
112      *      - testCreatePack is a void method, it creates packs of odd values given
113      *      to it and then check it's size, empties the pack
114      *      then it would move on to the next odd input for createPack method.
115      *
116      * @link CardGame.java
117      *
118      * @CardGameClassInstance cardGame
119      * @CardGameClassMethods createPack(int), emptyPack(), getPack()
120      */
121     @Test
122     public void testCreatePack() {
123 
124         try {
125             CardGame cardGame = new CardGame();
126             Class<?> cardGameClass = cardGame.getClass();
127             Method emptyPackMethod = cardGameClass.getDeclaredMethod("emptyPack");
128             Method createPackMethod = cardGameClass.getDeclaredMethod("createPack", int.class);
129             Method getPackMethod = cardGameClass.getDeclaredMethod("getPack");
130 
131             emptyPackMethod.setAccessible(true);
132             createPackMethod.setAccessible(true);
133             getPackMethod.setAccessible(true);
134 
135             emptyPackMethod.invoke(cardGame);
136             createPackMethod.invoke(cardGame, 1);
137             assertEquals(8, ((LinkedList<Integer>) getPackMethod.invoke(cardGame)).size());
138 
139             emptyPackMethod.invoke(cardGame);
140             createPackMethod.invoke(cardGame, 0);
141             assertEquals(8, ((LinkedList<Integer>) getPackMethod.invoke(cardGame)).size());
142 
143             emptyPackMethod.invoke(cardGame);
144             createPackMethod.invoke(cardGame, -1);
145             assertEquals(8, ((LinkedList<Integer>) getPackMethod.invoke(cardGame)).size());
146 
147             emptyPackMethod.invoke(cardGame);
148 
149             createPackMethod.invoke(cardGame, -1001);
150             assertEquals(8, ((LinkedList<Integer>) getPackMethod.invoke(cardGame)).size());
151         } catch (Exception e) {
152             fail("testCreatePack Failed");
153         }
154     }
155 
156     /**
157      * @see testGet1FromPack
158      *
159      *      - testGet1FromPack is a void method, it creates a pack and retrieves the
```

```
160     *      first element of the pack
161     *      then it would use get1FromPack method to retrieve and remove the first
162     *      element of the pack, after doing so
163     *      the code checks to see if the correct card was retrieved from this
164     *      method and if the size is reduced by 1.
165     *
166     * @link CardGame.java
167     *
168     * @CardGameClassInstance cardGame
169     * @CardGameClassMethods createPack(int), get1FromPack(), getPack()
170     */
171    @Test
172    public void testGet1FromPack() {
173
174        try {
175            CardGame cardGame = new CardGame();
176            Class<?> cardGameClass = cardGame.getClass();
177            Method createPackMethod = cardGameClass.getDeclaredMethod("createPack", int.class);
178            Method getPackMethod = cardGameClass.getDeclaredMethod("getPack");
179            Method get1FromPackMethod = cardGameClass.getDeclaredMethod("get1FromPack");
180
181            createPackMethod.setAccessible(true);
182            getPackMethod.setAccessible(true);
183            get1FromPackMethod.setAccessible(true);
184
185            createPackMethod.invoke(cardGame, 1);
186
187            int packSize = ((LinkedList<Integer>) getPackMethod.invoke(cardGame)).size();
188            int expected = ((LinkedList<Integer>) getPackMethod.invoke(cardGame)).getFirst();
189            int actual = ((int) get1FromPackMethod.invoke(cardGame));
190
191            assertEquals(expected, actual);
192            assertEquals(packSize - 1, ((LinkedList<Integer>) getPackMethod.invoke(cardGame)).size());
193        } catch (Exception e) {
194            fail("testGet1FromPack Failed");
195        }
196    }
197
198    /**
199     * @see testSetPlayers
200     *
201     *      - testSetPlayers is a void method, it creates 6 empty lists within the
202     *      nested linked lists called players, then it would retrieve this list
203     *      from Card class and after creating a local nested list
204     *      of the same length, it would compare to see if they are the same lists.
205     *
206     * @link CardGame.java
207     *
208     * @CardGameClassInstance cardGame
209     * @CardGameClassMethods setPlayers(), getPlayers()
210     */
211    @Test
212    public void testSetPlayers() {
213        try {
```

```
214             CardGame cardGame = new CardGame();
215             Class<?> cardGameClass = cardGame.getClass();
216             Method setPlayersMethod = cardGameClass.getDeclaredMethod("setPlayers", int.class);
217             Method getPlayersMethod = cardGameClass.getDeclaredMethod("getPlayers");
218 
219             setPlayersMethod.setAccessible(true);
220             getPlayersMethod.setAccessible(true);
221 
222             setPlayersMethod.invoke(cardGame, 6);
223 
224             LinkedList<LinkedList<Integer>> actual = (LinkedList<LinkedList<Integer>>) getPlayersMethod
225                     .invoke(cardGame);
226             LinkedList<LinkedList<Integer>> expected = new LinkedList<LinkedList<Integer>>();
227 
228             for (int i = 0; i < 6; i++) {
229                 expected.add(new LinkedList<Integer>());
230             }
231 
232             assertEquals(6, actual.size());
233             assertEquals(expected, actual);
234 
235         } catch (Exception e) {
236             fail("testSetPlayers Failed");
237         }
238 
239     }
240 
241     /**
242      * @see testSetDecks
243      *
244      *      - testSetDecks is a void method, similarly to the testSetPlayers method
245      *      it would set the decks within the CardGame Class and retrieves it and
246      *      then
247      *      creates the
248      *      expected local list and compares the two lists for Equality and Size.
249      *
250      * @link CardGame.java
251      *
252      * @CardGameClassInstance cardGame
253      * @CardGameClassMethods setDecks(), getDecks()
254      */
255     @Test
256     public void testSetDecks() {
257 
258         try {
259             CardGame cardGame = new CardGame();
260             Class<?> cardGameClass = cardGame.getClass();
261             Method setDecksMethod = cardGameClass.getDeclaredMethod("setDecks", int.class);
262             Method getDecksMethod = cardGameClass.getDeclaredMethod("getDecks");
263 
264             setDecksMethod.setAccessible(true);
265             getDecksMethod.setAccessible(true);
266 
267             setDecksMethod.invoke(cardGame, 3);
```

```
268
269         LinkedList<LinkedList<Integer>> actual = (LinkedList<LinkedList<Integer>>) getDecksMethod
270                 .invoke(cardGame);
271         LinkedList<LinkedList<Integer>> expected = new LinkedList<LinkedList<Integer>>();
272
273         for (int i = 0; i < 3; i++) {
274            expected.add(new LinkedList<Integer>());
275         }
276
277         assertEquals(3, actual.size());
278         assertEquals(expected, actual);
279
280      } catch (Exception e) {
281         fail("testSetDecks Failed");
282      }
283   }
284
285   /**
286    * @see testDealHands
287    *
288    *      - testDealHands is a void method, it would create a pack for 5 players
289    *      of size 40, then it would retrieve the first 5 cards within the pack,
290    *      then it would use methods setPlayers and dealHands to distribute half of
291    *      the pack to the players then it would check that dealer has distributed
292    *      the cards in the correct format
293    *      giving one card at a time to each player in a round robin fashion, then
294    *      it would check that all players have 4 cards in their hands.
295    *
296    * @link CardGame.java
297    *
298    * @CardGameClassInstance cardGame
299    * @InstanceAttributes playerNumber
300    * @CardGameClassMethods createPack(int), setPlayers(int), getPlayers(),
301    *                       dealHands(), getPack()
302    */
303   @Test
304   public void testDealHands() {
305      try {
306         CardGame cardGame = new CardGame();
307         Class<?> cardGameClass = cardGame.getClass();
308         Method createPackMethod = cardGameClass.getDeclaredMethod("createPack", int.class);
309         Method getPackMethod = cardGameClass.getDeclaredMethod("getPack");
310         Method setPlayersMethod = cardGameClass.getDeclaredMethod("setPlayers", int.class);
311         Method dealHandsMethod = cardGameClass.getDeclaredMethod("dealHands");
312         Method getPlayersMethod = cardGameClass.getDeclaredMethod("getPlayers");
313         Field playerNumberField = cardGameClass.getDeclaredField("playerNumber");
314
315         createPackMethod.setAccessible(true);
316         getPackMethod.setAccessible(true);
317         playerNumberField.setAccessible(true);
318         setPlayersMethod.setAccessible(true);
319         dealHandsMethod.setAccessible(true);
320         getPlayersMethod.setAccessible(true);
321
```

```
322             playerNumberField.set(cardGame, 5);
323             createPackMethod.invoke(cardGame, (int) playerNumberField.get(cardGame));
324 
325             LinkedList<Integer> pack = (LinkedList<Integer>) getPackMethod.invoke(cardGame);
326 
327             int player1Card1 = pack.get(0);
328             int player2Card1 = pack.get(1);
329             int player3Card1 = pack.get(2);
330             int player4Card1 = pack.get(3);
331             int player5Card1 = pack.get(4);
332             int player1Card2 = pack.get(5);
333 
334             setPlayersMethod.invoke(cardGame, (int) playerNumberField.get(cardGame));
335 
336             LinkedList<LinkedList<Integer>> actual = (LinkedList<LinkedList<Integer>>)
    dealHandsMethod.invoke(cardGame);
337 
338             assertTrue(player1Card1 == ((LinkedList<LinkedList<Integer>>) getPlayersMethod.invoke(cardGame)).get(0)
339                     .get(0));
340             assertTrue(player2Card1 == ((LinkedList<LinkedList<Integer>>) getPlayersMethod.invoke(cardGame)).get(1)
341                     .get(0));
342             assertTrue(player3Card1 == ((LinkedList<LinkedList<Integer>>) getPlayersMethod.invoke(cardGame)).get(2)
343                     .get(0));
344             assertTrue(player4Card1 == ((LinkedList<LinkedList<Integer>>) getPlayersMethod.invoke(cardGame)).get(3)
345                     .get(0));
346             assertTrue(player5Card1 == ((LinkedList<LinkedList<Integer>>) getPlayersMethod.invoke(cardGame)).get(4)
347                     .get(0));
348             assertTrue(player1Card2 == ((LinkedList<LinkedList<Integer>>) getPlayersMethod.invoke(cardGame)).get(0)
349                     .get(1));
350 
351 
352             assertEquals(4, actual.get(0).size());
353             assertEquals(4, actual.get(1).size());
354             assertEquals(4, actual.get(2).size());
355             assertEquals(4, actual.get(3).size());
356             assertEquals(4, actual.get(4).size());
357 
358 
359             assertTrue(actual.size() == (int) playerNumberField.get(cardGame));
360         } catch (Exception e) {
361             fail("testDealHands Failed");
362         }
363     }
364 
365     /**
366      * @see testDealHandsMockHands
367      *
368      *      - testDealHandsMockHands is a void method, it would create a pack with
369      *      wrong values and size then
370      *      it would checks to see if the hands where created and that cards where
371      *      distributed correctly among players.
372      *
373      * @link CardGame.java
374      *
```

```
375      * @CardGameClassInstance cardGame
376      * @InstanceAttributes playerNumber, pack, players
377      * @CardGameClassMethods emptyPack(), setPlayers(), dealHands()
378      */
379     @Test
380     public void testDealHandsMockHands() {
381         try {
382             CardGame cardGame = new CardGame();
383             Class<?> cardGameClass = cardGame.getClass();
384             Method setPlayersMethod = cardGameClass.getDeclaredMethod("setPlayers", int.class);
385             Method dealHandsMethod = cardGameClass.getDeclaredMethod("dealHands");
386             Method emptyPackMethod = cardGameClass.getDeclaredMethod("emptyPack");
387             Field playerNumberField = cardGameClass.getDeclaredField("playerNumber");
388             Field packField = cardGameClass.getDeclaredField("pack");
389             Field playersField = cardGameClass.getDeclaredField("players");
390 
391             playerNumberField.setAccessible(true);
392             setPlayersMethod.setAccessible(true);
393             dealHandsMethod.setAccessible(true);
394             packField.setAccessible(true);
395             emptyPackMethod.setAccessible(true);
396             playersField.setAccessible(true);
397 
398             emptyPackMethod.invoke(cardGame);
399 
400             LinkedList<Integer> pack = new LinkedList<>();
401 
402             pack.add(9);
403 
404             pack.add(23);
405             pack.add(34);
406             pack.add(5);
407             pack.add(0);
408             pack.add(-23);
409             pack.add(-4);
410             pack.add(0);
411             pack.add(1);
412 
413             packField.set(cardGame, pack);
414 
415             playerNumberField.set(cardGame, 4);
416             setPlayersMethod.invoke(cardGame, (int) playerNumberField.get(cardGame));
417 
418             LinkedList<LinkedList<Integer>> mockHands = (LinkedList<LinkedList<Integer>>) dealHandsMethod
419                     .invoke(cardGame);
420             assertTrue(!((LinkedList<LinkedList<Integer>>) playersField.get(cardGame)).getFirst().isEmpty());
421 
422             // since the number of elements added to the pack is 9
423             boolean expected = (mockHands.get(0).size() + mockHands.get(1).size() + mockHands.get(2).size()
424                     + mockHands.get(3).size()) == 9;
425 
426             assertTrue(expected);
427 
428         } catch (Exception e) {
```

```
429             fail("testDealHandsMockHands Failed");
430         }
431     }
432
433     /**
434      * @see testDealDecks
435      *
436      *      - testDealDecks is a void method, it creates a pack and distributes it's
437      *      card to players and decks and it
438      *      would check to see if all nested decks have the same size 4 and check to
439      *      see if the decks are valid and pack is empty.
440      *
441      * @Note on successful run we know that we have n decks of size 4 and from the
442      *       previous test we have n hands of size 4
443      *       therefore we have total cards of 8 * n which is the same as
444      *       specification.
445      *
446      * @link CardGame.java
447      *
448      * @CardGameClassInstance cardGame
449      * @InstanceAttributes playerNumber
450      * @CardGameClassMethods createPack(int), setPlayers(), dealHands(),
451      *                       dealDecks(),
452      *                       seDecks(), getPack()
453      */
454     @Test
455     public void testDealDecks() {
456         try {
457             CardGame cardGame = new CardGame();
458             Class<?> cardGameClass = cardGame.getClass();
459             Method createPackMethod = cardGameClass.getDeclaredMethod("createPack", int.class);
460             Method getPackMethod = cardGameClass.getDeclaredMethod("getPack");
461             Method setPlayersMethod = cardGameClass.getDeclaredMethod("setPlayers", int.class);
462             Method dealHandsMethod = cardGameClass.getDeclaredMethod("dealHands");
463             Method dealDecksMethod = cardGameClass.getDeclaredMethod("dealDecks");
464             Field playerNumberField = cardGameClass.getDeclaredField("playerNumber");
465             Method setDecksMethod = cardGameClass.getDeclaredMethod("setDecks", int.class);
466             Method getDecksMethod = cardGameClass.getDeclaredMethod("getDecks");
467
468             setDecksMethod.setAccessible(true);
469             getDecksMethod.setAccessible(true);
470             createPackMethod.setAccessible(true);
471             getPackMethod.setAccessible(true);
472             playerNumberField.setAccessible(true);
473             setPlayersMethod.setAccessible(true);
474             dealHandsMethod.setAccessible(true);
475             dealDecksMethod.setAccessible(true);
476
477             playerNumberField.set(cardGame, 4);
478             createPackMethod.invoke(cardGame, (int) playerNumberField.get(cardGame));
479
480             LinkedList<Integer> pack = (LinkedList<Integer>) getPackMethod.invoke(cardGame);
481
482             setPlayersMethod.invoke(cardGame, (int) playerNumberField.get(cardGame));
```

```
483             dealHandsMethod.invoke(cardGame);
484
485             setDecksMethod.invoke(cardGame, (int) playerNumberField.get(cardGame));
486
487             LinkedList<LinkedList<Integer>> actual = (LinkedList<LinkedList<Integer>>)
    dealDecksMethod.invoke(cardGame);
488
489             assertEquals(4, actual.get(0).size());
490             assertTrue(actual.get(1).size() == 4);
491             assertTrue(actual.get(2).size() == 4);
492             assertTrue(actual.get(3).size() == 4);
493             assertTrue(actual.size() == (int) playerNumberField.get(cardGame));
494             assertTrue(pack.size() == 0);
495
496         } catch (Exception e) {
497             fail("testDealDecks Failed");
498         }
499     }
500
501     /**
502      * @see testDealDecksMockHands
503      *
504      *      - testDealDecksMockHands is a void method, it creates a pack with wrong
505      *      values and just distributes the cards of this pack
506      *      between decks then it would check if the pack has been created and if it
507      *      contains all the values inside the pack and that the pack is empty.
508      *
509      * @link CardGame.java
510      *
511      * @CardGameClassInstance cardGame
512      * @InstanceAttributes playerNumber, pack
513      * @CardGameClassMethods dealDecks(), seDecks(), emptyPack()
514      */
515     @Test
516     public void testDealDecksMockHands() {
517         try {
518             CardGame cardGame = new CardGame();
519             Class<?> cardGameClass = cardGame.getClass();
520             Method setDecksMethod = cardGameClass.getDeclaredMethod("setDecks", int.class);
521             Method dealDecksMethod = cardGameClass.getDeclaredMethod("dealDecks");
522             Method emptyPackMethod = cardGameClass.getDeclaredMethod("emptyPack");
523             Field playerNumberField = cardGameClass.getDeclaredField("playerNumber");
524             Field packField = cardGameClass.getDeclaredField("pack");
525             Field decksField = cardGameClass.getDeclaredField("decks");
526
527             playerNumberField.setAccessible(true);
528             setDecksMethod.setAccessible(true);
529             dealDecksMethod.setAccessible(true);
530             packField.setAccessible(true);
531             emptyPackMethod.setAccessible(true);
532             decksField.setAccessible(true);
533
534             emptyPackMethod.invoke(cardGame);
535
```

```
536             LinkedList<Integer> pack = new LinkedList<>();
537
538             pack.add(3);
539             pack.add(213);
540             pack.add(400);
541             pack.add(5 - 12);
542             pack.add(000);
543             pack.add(-234454353);
544             pack.add(-23843);
545             pack.add(-000000000);
546             pack.add(134443);
547
548             packField.set(cardGame, pack);
549
550             playerNumberField.set(cardGame, 4);
551             setDecksMethod.invoke(cardGame, (int) playerNumberField.get(cardGame));
552
553             LinkedList<LinkedList<Integer>> mockHands = (LinkedList<LinkedList<Integer>>) dealDecksMethod
554                     .invoke(cardGame);
555             assertTrue(!((LinkedList<LinkedList<Integer>>) decksField.get(cardGame)).getFirst().isEmpty());
556             assertTrue(((LinkedList<Integer>) packField.get(cardGame)).isEmpty());
557
558             // since the number of elements added to the pack is 9
559             boolean expected = (mockHands.get(0).size() + mockHands.get(1).size() + mockHands.get(2).size()
560                     + mockHands.get(3).size()) == 9;
561
562             assertTrue(expected);
563
564         } catch (Exception e) {
565             fail("testDealDecksMockHands Failed");
566         }
567     }
568
569     /**
570      * @see testGetPlayers
571      *
572      *      - testGetPlayers is a void method, it checks the functionality of
573      *      getPlayers by adding some elements to the main players list within
574      *      CardGame
575      *      Class,
576      *      and checking the validity of the players afterwards.
577      *
578      * @link CardGame.java
579      *
580      * @CardGameClassInstance cardGame
581      * @InstanceAttributes players
582      * @CardGameClassMethods setPlayers(), getPlayers()
583      */
584     @Test
585     public void testGetPlayers() {
586         try {
587             CardGame cardGame = new CardGame();
588             Class<?> cardGameClass = cardGame.getClass();
589             Method setPlayersMethod = cardGameClass.getDeclaredMethod("setPlayers", int.class);
```

```
590             Method getPlayersMethod = cardGameClass.getDeclaredMethod("getPlayers");
591             Field playersField = cardGameClass.getDeclaredField("players");
592 
593             setPlayersMethod.setAccessible(true);
594             getPlayersMethod.setAccessible(true);
595             playersField.setAccessible(true);
596 
597             setPlayersMethod.invoke(cardGame, 5);
598 
599             ((LinkedList<LinkedList<Integer>>) playersField.get(cardGame)).get(0).add(90);
600             ((LinkedList<LinkedList<Integer>>) playersField.get(cardGame)).getLast().add(91);
601 
602             assertTrue(((LinkedList<LinkedList<Integer>>) playersField.get(cardGame)).getFirst().contains(90)
603                     && ((LinkedList<LinkedList<Integer>>) playersField.get(cardGame)).getLast().contains(91));
604         } catch (Exception e) {
605             fail("testGetPlayers Failed");
606         }
607     }
608 
609     /**
610      * @see testGetDecks
611      *
612      *      - testGetDecks is a void method, it checks the functionality of getDecks
613      *      by adding some elements to the main decks list within Card Class,
614      *      and checking the validity of the decks afterwards.
615      *
616      * @link CardGame.java
617      *
618      * @CardGameClassInstance cardGame
619      * @InstanceAttributes decks
620      * @CardGameClassMethods setDecks(), getDecks()
621      */
622     @Test
623     public void testGetDecks() {
624         try {
625             CardGame cardGame = new CardGame();
626             Class<?> cardGameClass = cardGame.getClass();
627             Method setPlayersMethod = cardGameClass.getDeclaredMethod("setDecks", int.class);
628             Method getPlayersMethod = cardGameClass.getDeclaredMethod("getDecks");
629             Field decksField = cardGameClass.getDeclaredField("decks");
630 
631             setPlayersMethod.setAccessible(true);
632             getPlayersMethod.setAccessible(true);
633             decksField.setAccessible(true);
634 
635             setPlayersMethod.invoke(cardGame, 7);
636 
637             ((LinkedList<LinkedList<Integer>>) decksField.get(cardGame)).get(0).add(100);
638             ((LinkedList<LinkedList<Integer>>) decksField.get(cardGame)).getLast().add(101);
639 
640             assertTrue(((LinkedList<LinkedList<Integer>>) decksField.get(cardGame)).getFirst().contains(100)
641                     && ((LinkedList<LinkedList<Integer>>) decksField.get(cardGame)).getLast().contains(101));
642         } catch (Exception e) {
643             fail("testGetDecks Failed");
```

```
644         }
645     }
646
647     /**
648      * @see testCreatePackOverload
649      *
650      *      - testCreatePackOverload is a void method, this method attempts to
651      *      overload the pack created by the system.
652      *
653      * @link CardGame.java
654      *
655      * @CardGameClassInstance cardGame
656      * @InstanceAttributes playerNumber
657      * @CardGameClassMethods createPack(int), getPack()
658      */
659     @BeforeClass // it requires the cache to be full for this test
660     public static void testCreatePackOverload() {
661         try {
662             CardGame cardGame = new CardGame();
663             Class<?> cardGameClass = cardGame.getClass();
664             Method createPackMethod = cardGameClass.getDeclaredMethod("createPack", int.class);
665             Method getPackMethod = cardGameClass.getDeclaredMethod("getPack");
666             Field playerNumberField = cardGameClass.getDeclaredField("playerNumber");
667
668             createPackMethod.setAccessible(true);
669             playerNumberField.setAccessible(true);
670             getPackMethod.setAccessible(true);
671
672             playerNumberField.set(cardGame, 2147483647); // passing the largest integer possible
673             createPackMethod.invoke(cardGame, (int) playerNumberField.get(cardGame));
674             assertEquals(0, ((LinkedList<Integer>) getPackMethod.invoke(cardGame)).size());
675
676             playerNumberField.set(cardGame, 21474839 / 16);// maximum value executable by the method
677             // tolerance (or delta) is because of the integer divisions might return a
678             // result not as exact
679             createPackMethod.invoke(cardGame, (int) playerNumberField.get(cardGame));
680             assertEquals(21474839 / 2, ((LinkedList<Integer>) getPackMethod.invoke(cardGame)).size(), 3);
681
682         } catch (Exception e) {
683             fail("testCreatePackOverload Failed");
684         }
685     }
686
687     /**
688      * @see testCardConstructor
689      *
690      *      - testCardConstructor is a void method, this method attempt to test the
691      *      constructor of the CardGame class, it would pass wrong values
692      *      as player number and overloads the constructor with 100000 players.
693      *
694      * @Note when this constructor is called it would set player numbers, creates a
695      *       pack, sets both players and decks main list then it would
696      *       distribute the cards within that pack among players and decks in a
697      *       round robin fashion.
```

```
698      *
699      * @link CardGame.java
700      *
701      * @CardGameClassInstance cardGame, cardGame1, cardGame2, cardGame3, cardGame4
702      * @CardClassMethods getDecks()
703      */
704
705     @BeforeClass
706     public static void testCardGameConstructor() {
707
708         try {
709             CardGame cardGame = new CardGame();
710             Class<?> cardGameClass = cardGame.getClass();
711
712             Method createPackMethod = cardGameClass.getDeclaredMethod("createPack", int.class);
713             createPackMethod.setAccessible(true);
714
715             LinkedList<Integer> pack = (LinkedList<Integer>) createPackMethod.invoke(cardGame, -74);
716
717             CardGame cardGame1 = new CardGame(-74, pack);
718
719             Class<?> cardGameClass1 = cardGame1.getClass();
720             Method getPlayersMethod = cardGameClass1.getDeclaredMethod("getPlayers");
721             getPlayersMethod.setAccessible(true);
722
723             assertEquals(1, ((LinkedList<LinkedList<Integer>>) getPlayersMethod.invoke(cardGame1)).size());
724
725             pack = (LinkedList<Integer>) createPackMethod.invoke(cardGame, 0);
726             CardGame cardGame2 = new CardGame(0, pack);
727
728             Class<?> cardGameClass2 = cardGame2.getClass();
729             Method getPlayersMethod2 = cardGameClass2.getDeclaredMethod("getPlayers");
730             getPlayersMethod2.setAccessible(true);
731
732             assertEquals(1, ((LinkedList<LinkedList<Integer>>) getPlayersMethod2.invoke(cardGame2)).size());
733
734             // it even works with 100000 players but due to the system overload for the
735             // players and deck files created it would crash the system
736             pack = (LinkedList<Integer>) createPackMethod.invoke(cardGame, 1000);
737             CardGame cardGame3 = new CardGame(1000, pack);
738
739             Class<?> cardGameClass3 = cardGame3.getClass();
740             Method getPlayersMethod3 = cardGameClass3.getDeclaredMethod("getPlayers");
741             getPlayersMethod3.setAccessible(true);
742
743             assertEquals(1000, ((LinkedList<LinkedList<Integer>>) getPlayersMethod3.invoke(cardGame3)).size());
744
745             for (int i = 1; i < 11; i++) {
746                 pack = (LinkedList<Integer>) createPackMethod.invoke(cardGame, i);
747                 CardGame cardGame4 = new CardGame(i, pack);
748
749                 Class<?> cardGameClass4 = cardGame4.getClass();
750                 Method getDecksMethod = cardGameClass4.getDeclaredMethod("getDecks");
751                 getDecksMethod.setAccessible(true);
```

```
752
753             assertEquals(i, ((LinkedList<LinkedList<Integer>>) getDecksMethod.invoke(cardGame4)).size());
754         }
755
756     } catch (Exception e) {
757         fail("testCardGameConstructor Failed");
758     }
759 }
760
761 /**
762  * @see testCardGameGameStability
763  *
764  *      - testCardGameGameStability is a void method, this method will attempt
765  *      to start the main game method with an arbitrary pre set hand and deck
766  *      which should win the game on the spot, then it tries to modify that hand
767  *      while the game is being played.
768  *
769  * @Note if the number of players change during a game the program will print an
770  *       IndexOutOfBounds exception but the game continues.
771  *
772  * @link Player.java, Card.java, CardGame.java
773  *
774  * @CardGameClassInstance cardGame
775  * @InstanceAttributes playerNumber
776  * @CardGameClassMethods setPlayers(), setDecks(), getPlayers(), getDecks(),
777  *                       startGame()
778  *
779  * @PlayerClassInstance player
780  * @PlayerClassMethods Player.getPlayers()
781  *
782  * @CardClassInstance card
783  * @CardClassMethods Card.getDecks()
784  */
785 @BeforeClass
786 public static void testCardGameGameStability() {
787
788     try {
789         CardGame cardGame = new CardGame();
790         Class<?> cardGameClass = cardGame.getClass();
791         Method getPackMethod = cardGameClass.getDeclaredMethod("getPack");
792         Method setPlayersMethod = cardGameClass.getDeclaredMethod("setPlayers", int.class);
793         Method setDecksMethod = cardGameClass.getDeclaredMethod("setDecks", int.class);
794         Method getPlayersMethod = cardGameClass.getDeclaredMethod("getPlayers");
795         Method getDecksMethod = cardGameClass.getDeclaredMethod("getDecks");
796         Method startGameMethod = cardGameClass.getDeclaredMethod("startGame");
797         Field playerNumberField = cardGameClass.getDeclaredField("playerNumber");
798
799         playerNumberField.setAccessible(true);
800         getPackMethod.setAccessible(true);
801         setDecksMethod.setAccessible(true);
802         setPlayersMethod.setAccessible(true);
803         getDecksMethod.setAccessible(true);
804         getPlayersMethod.setAccessible(true);
805         startGameMethod.setAccessible(true);
```

```
806
807             playerNumberField.set(cardGame, 8);
808             setPlayersMethod.invoke(cardGame, 8);
809             setDecksMethod.invoke(cardGame, 8);
810
811             Player player = new Player(((LinkedList<LinkedList<Integer>>) getPlayersMethod.invoke(cardGame)));
812             Card card = new Card((((LinkedList<LinkedList<Integer>>) getDecksMethod.invoke(cardGame))));
813
814             Player.getPlayers().getFirst().add(1);
815             Player.getPlayers().getFirst().add(1);
816             Player.getPlayers().getFirst().add(1);
817             Player.getPlayers().getFirst().add(0);
818
819             Card.getDecks().getFirst().add(0);
820             Card.getDecks().getFirst().add(0);
821             Card.getDecks().getFirst().add(0);
822             Card.getDecks().getFirst().add(1);
823
824             startGameMethod.invoke(cardGame);
825
826             playerNumberField.set(cardGame, 8);
827             setPlayersMethod.invoke(cardGame, 8);
828             setDecksMethod.invoke(cardGame, 8);
829
830             Player newPlayer = new Player(((LinkedList<LinkedList<Integer>>) getPlayersMethod.invoke(cardGame)));
831             Card newCard = new Card((((LinkedList<LinkedList<Integer>>) getDecksMethod.invoke(cardGame))));
832
833             Player.getPlayers().get(0).add(13);
834             Player.getPlayers().get(0).add(16);
835             Player.getPlayers().get(0).add(5);
836             Player.getPlayers().get(0).add(2);
837             Player.getPlayers().get(0).add(3);
838             Player.getPlayers().get(0).add(13);
839             Player.getPlayers().get(1).add(3);
840             Player.getPlayers().get(2).add(3);
841             Player.getPlayers().get(2).add(2);
842             Player.getPlayers().get(2).add(1);
843             Player.getPlayers().get(2).add(4);
844             Player.getPlayers().get(2).add(4);
845             Player.getPlayers().get(2).add(3);
846             Player.getPlayers().get(2).add(3);
847             Player.getPlayers().get(2).add(3);
848
849             Card.getDecks().get(0).add(3);
850             Card.getDecks().get(1).add(3);
851             Card.getDecks().get(1).add(3);
852             Card.getDecks().get(2).add(3);
853             Card.getDecks().get(2).add(3);
854             Card.getDecks().get(2).add(3);
855             Card.getDecks().get(2).add(3);
856             Card.getDecks().get(2).add(3);
857             Card.getDecks().get(2).add(3);
858
859             assertTrue(true); // game starts successfully
```

```
860
861         } catch (Exception e) {
862             fail("testCardGameGameStability Failed");
863         }
864     }
865
866     /**
867      * @see testStartGameOverload
868      *
869      *      - testStartGameOverload is a void method, this method will attempt to
870      *      start a game with 2000 players and successfully declare a winner.
871      *
872      * @link Player.java, Card.java
873      *
874      * @PlayerClassInstance player
875      * @PlayerClassMethods startGame()
876      *
877      * @CardClassInstance card
878      * @CardClassMethods getPlayers(), getDecks()
879      */
880     @BeforeClass
881     public static void testStartGameOverload() {
882         try {
883             CardGame cardGame = new CardGame();
884             Class<?> cardGameClass = cardGame.getClass();
885             Method getPlayersMethod = cardGameClass.getDeclaredMethod("getPlayers");
886             Method getDecksMethod = cardGameClass.getDeclaredMethod("getDecks");
887             Method createPackMethod = cardGameClass.getDeclaredMethod("createPack", int.class);
888             Method setPlayersMethod = cardGameClass.getDeclaredMethod("setPlayers", int.class);
889             Method setDecksMethod = cardGameClass.getDeclaredMethod("setDecks", int.class);
890             Method dealHandsMethod = cardGameClass.getDeclaredMethod("dealHands");
891             Method dealDecksMethod = cardGameClass.getDeclaredMethod("dealDecks");
892             Method startGameMethod = cardGameClass.getDeclaredMethod("startGame");
893             Field playerNumberField = cardGameClass.getDeclaredField("playerNumber");
894
895             playerNumberField.setAccessible(true);
896             getDecksMethod.setAccessible(true);
897             getPlayersMethod.setAccessible(true);
898             setDecksMethod.setAccessible(true);
899             setPlayersMethod.setAccessible(true);
900             dealDecksMethod.setAccessible(true);
901             dealHandsMethod.setAccessible(true);
902             startGameMethod.setAccessible(true);
903             createPackMethod.setAccessible(true);
904
905             playerNumberField.set(cardGame, 2000);
906             LinkedList<Integer> pack = (LinkedList<Integer>) createPackMethod.invoke(cardGame, 8000);
907             setPlayersMethod.invoke(cardGame, (int) playerNumberField.get(cardGame));
908             setDecksMethod.invoke(cardGame, (int) playerNumberField.get(cardGame));
909
910             InputOutput output = new InputOutput((LinkedList<LinkedList<Integer>>)
dealHandsMethod.invoke(cardGame));
911             dealDecksMethod.invoke(cardGame);
```

```
912
913             Player player = new Player((LinkedList<LinkedList<Integer>>) getPlayersMethod.invoke(cardGame));
914             Card card = new Card((LinkedList<LinkedList<Integer>>) getDecksMethod.invoke(cardGame));
915
916             startGameMethod.invoke(cardGame);
917             assertTrue(true); // game starts successfully with no errors
918
919         } catch (Exception e) {
920             fail("testStartGameOverload Failed");
921         }
922     }
923 }
924
```

testCard.java

```
1
2
3  import static org.junit.Assert.assertEquals;
4  import static org.junit.Assert.assertTrue;
5  import java.util.LinkedList;
6  import org.junit.Test;
7
8  /**
9   * @see testCard
10  *
11  *      - Class testCard is used for testing all the methods and objects as well
12  *      as the constructors of the
13  *      Card Class, It checks the code for certain exceptions that could occur.
14  *
15  * @Note most test methods within this test class are testing multiple aspects
16  *       of the class such as other methods and objects.
17  *
18  * @Note getDecks method's functionality is tested within other test methods
19  *       thus it doesn't have it's own test method.
20  *
21  * @author Amirali Famili
22  */
23 public class testCard {
24
25     /**
26      * @see testCardConstructor
27      *
28      *      - testCardConstructor is a void method, it tests the constructor of
29      *      the Card class to make sure that decks LinkedList and the number of
30      *      decks (or players) are set.
31      *
32      * @link Card.java
33      *
34      * @CardClassInstance card
35      * @InstanceAttributes deckNumber
36      * @PlayerClassMethods getDecks()
37      */
38     @Test
39     public void testCardConstructor() {
40         LinkedList<LinkedList<Integer>> decks = new LinkedList<>();
41         for (int i = 0; i < 3; i++) {
42             decks.add(new LinkedList<>());
43             for (int j = 0; j < 2; j++) {
44                 decks.get(i).add(j);
45             }
46         }
47         Card card = new Card(decks);
48
49
50         assertEquals(3, Card.getDecks().size());
51         assertEquals(2, Card.getDecks().get(0).size());
```

```
52          assertEquals(2, Card.getDecks().get(1).size());
53          assertEquals(2, Card.getDecks().get(2).size());
54          assertEquals(3, card.deckNumber);
55      }
56
57      /**
58       * @see testGetLeftDeck
59       *
60       *      - testGetLeftDeck is a void method, it tests the getLeftDeck method from
61       *      Card class by creating a mock decks LinkedList and assigning a
62       *      playerIndex, by passing the player index to the method we get their left
63       *      deck.
64       *
65       * @Note player always has the same index as their left deck.
66       *
67       * @link Card.java
68       *
69       * @CardClassInstance card
70       * @InstanceAttributes playerNumber
71       * @PlayerClassMethods getLeftDeck()
72       */
73      @Test
74      public void testGetLeftDeck() {
75
76          LinkedList<LinkedList<Integer>> decks = new LinkedList<>();
77          for (int i = 0; i < 3; i++) {
78              decks.add(new LinkedList<>());
79          }
80
81          decks.get(0).add(null);
82          decks.get(0).add(1);
83          decks.get(0).add(0);
84          decks.get(1).add(null);
85          decks.get(1).add(0);
86          decks.get(1).add(0);
87          decks.get(1).add(0);
88          decks.get(1).add(0);
89
90          Card card = new Card(decks);
91
92          int playerIndex = 0;
93
94          assertEquals(3, card.getLeftDeck(playerIndex).size());
95
96          playerIndex++;
97
98          assertEquals(5, card.getLeftDeck(playerIndex).size());
99
100         assertEquals(3, card.getLeftDeck(-1).size());
101         assertEquals(3, card.getLeftDeck(-2).size());
102         assertEquals(3, card.getLeftDeck(-3).size());
103         assertEquals(3, card.getLeftDeck(-4).size());
104         assertEquals(3, card.getLeftDeck(-5).size());
105     }
```

```
106
107     /**
108      * @see testGetRightDeck
109      *
110      *      - testGetRightDeck is a void method, it tests the getLeftDeck method
111      *      from
112      *      Card class by creating a mock decks LinkedList and assigning a
113      *      playerIndex, by passing the player index to the method we get their
114      *      right
115      *      deck, since we give it the last player's index it should return their
116      *      right deck which is the same as first player's left deck.
117      *
118      * @Note players right deck's index is always one index above their left deck.
119      *
120      * @link Card.java
121      *
122      * @CardClassInstance card
123      * @InstanceAttributes playerNumber
124      * @PlayerClassMethods getRightDeck()
125      */
126     @Test
127     public void testGetRightDeck() {
128         LinkedList<LinkedList<Integer>> decks = new LinkedList<>();
129         for (int i = 0; i < 3; i++) {
130             decks.add(new LinkedList<>());
131         }
132
133         decks.get(0).add(null);
134         decks.get(0).add(1);
135         decks.get(0).add(0);
136         decks.get(1).add(null);
137         decks.get(1).add(0);
138         decks.get(1).add(0);
139         decks.get(1).add(0);
140         decks.get(1).add(0);
141
142         Card card = new Card(decks);
143
144         // here we have 3 players and index 2 should point to the last player
145         // since last player's right deck is the same as left deck of the first player
146         int playerIndex = 2;
147         // this should point to the deck at index 0 since : 2+1 % 3 == 0
148         assertEquals(3, card.getRightDeck(playerIndex).size());
149
150         playerIndex++;
151
152         assertEquals(5, card.getRightDeck(playerIndex).size());
153
154         assertEquals(5, card.getRightDeck(-1).size());
155         assertEquals(5, card.getRightDeck(-2).size());
156         assertEquals(5, card.getRightDeck(-3).size());
157         assertEquals(5, card.getRightDeck(-4).size());
158         assertEquals(5, card.getRightDeck(-5).size());
159     }
```

```
160
161     /**
162      * @see testGetCardFromLeftDeck
163      *
164      *      - testGetCardFromLeftDeck is a void method, it tests the
165      *      getCardFromLeftDeck method from Card class, this method uses a poll
166      *      request to simply retrieve and remove the first element of it's list,
167      *      this method will check the functionality of the method.
168      *
169      * @Note when there is a null element method getCardFromLeftDeck will remove it
170      *       from the LinkedList and instead it would return -1 so that there
171      *       wouldn't be any interruption within the game.
172      *
173      * @link Card.java
174      *
175      * @CardClassInstance card
176      * @PlayerClassMethods getCardFromLeftDeck(LinkedList<Integer>)
177      */
178     @Test
179     public void testGetCardFromLeftDeck() {
180
181         Card card = new Card();
182
183         LinkedList<Integer> leftDeck = new LinkedList<Integer>();
184         leftDeck.add(null);
185         leftDeck.add(0);
186         leftDeck.add(1);
187         leftDeck.add(2);
188         leftDeck.add(-3748);
189
190         int drawnCard = card.getCardFromLeftDeck(leftDeck);
191
192         assertEquals(-1, drawnCard);
193
194         assertEquals(0, card.getCardFromLeftDeck(leftDeck));
195
196         assertEquals(-1, card.getCardFromLeftDeck(null));
197
198         assertEquals(-1, card.getCardFromLeftDeck(new LinkedList<>()));
199
200     }
201
202     /**
203      * @see testPutCardToRightDeck
204      *
205      *      - testPutCardToRightDeck is a void method, it tests the
206      *      putCardToRightDeck method from Card class, this method will receive an
207      *      integer and a list, then it would simple add the integer to the end of
208      *      the list, this test method will check the functionality and robustness of the method.
209      *
210      * @link Card.java
211      *
212      * @CardClassInstance card
213      * @PlayerClassMethods getCardFromLeftDeck(LinkedList<Integer>)
```

```
214      */
215     @Test
216     public void testPutCardToRightDeck() {
217
218         Card card = new Card();
219
220         LinkedList<Integer> leftDeck = new LinkedList<Integer>();
221         leftDeck.add(null);
222         leftDeck.add(0);
223         leftDeck.add(-1);
224         leftDeck.add(2);
225         leftDeck.add(-3748);
226
227         card.putCardToRightDeck(348, leftDeck);
228         int actual = leftDeck.getLast();
229         assertEquals(348, actual);
230
231         card.putCardToRightDeck(5, null);
232         int last = leftDeck.getLast();
233         assertEquals(348, last);
234
235         leftDeck = new LinkedList<>();
236         card.putCardToRightDeck(last, leftDeck);
237         assertEquals(1, leftDeck.size());
238         assertTrue(leftDeck.contains(348));
239     }
240 }
241
```

testPlayer.java

```java


import java.lang.reflect.Field;
import java.util.LinkedList;
import org.junit.Test;
import static org.junit.Assert.*;


/**
 * @see testPlayer
 *
 *      - Class testPlayer is used for testing all the methods and objects as
 *      well
 *      as the constructors of the
 *      Player Class, It checks the code for certain exceptions that could
 *      occur.
 *
 * @Note that most methods within this test class are testing multiple aspects
 *       of both the corresponding method as well as other methods
 *       which are consists of many assertEquals, assertTrue, ...
 *
 * @Note getPlayers functionality is already tested within some test methods
 *       thus I didn't test it in it's separate method.
 *
 * @author Amirali Famili
 */
public class testPlayer {

    /**
     * @see testPlayerConstructor
     *
     *      - testPlayerConstructor is a void method, it tests the constructor of
     *      the Player class to make sure that players LinkedList and the number of
     *      players are set.
     *
     * @link Player.java
     *
     * @PlayerClassInstance player
     * @InstanceAttributes playerNumber
     * @PlayerClassMethods getPlayers(), getPlayer(int)
     */
    @Test
    public void testPlayerConstructor() {
        LinkedList<LinkedList<Integer>> players = new LinkedList<>();
        for (int i = 0; i < 5; i++) {
            players.add(new LinkedList<>());
            for (int j = 0; j < 6; j++) {
                players.get(i).add(j);
            }
        }
        Player player = new Player(players);
```

```
52
53          assertEquals(5, Player.getPlayers().size());
54          assertEquals(6, player.getPlayer(0).size());
55          assertEquals(6, player.getPlayer(1).size());
56          assertEquals(6, player.getPlayer(2).size());
57          assertEquals(6, player.getPlayer(4).size());
58          assertEquals(5, player.playerNumber);
59      }
60
61      /**
62       * @see testReplaceCard
63       *
64       *      - testReplaceCard is a void method, it tests the replaceCard method
65       *      inside Players class by passing wrong and null values to it and it
66       *      checks the functionality of the method by creating a mock hand and
67       *      replacing a random value from it.
68       *
69       * @link Player.java
70       *
71       * @PlayerClassInstance player
72       * @PlayerClassMethods replaceCard(int, int, LinkedList<Integer>)
73       */
74      @Test
75      public void testReplaceCard() {
76          Player player = new Player();
77
78          try {
79              player.replaceCard(0, 0, null);
80              player.replaceCard(-3489, -783, null);
81              LinkedList<Integer> hand = new LinkedList<>();
82              hand.add(null);
83              hand.add(4);
84              hand.add(7);
85              hand.add(-348);
86              hand.add(0);
87              hand.add(0);
88              player.replaceCard(-8374, 1, hand);
89              player.replaceCard(748, 1, hand);
90              player.replaceCard(0, 1, hand);
91
92              assertTrue(hand.contains(1)); // 0 has been successfully replaced by 1
93          } catch (Exception e) {
94              fail("test failed at testReplaceCard, an error occurred whilst testing the method");
95          }
96      }
97
98      /**
99       * @see testGetPlayer
100      *
101      *      - testGetPlayer is a void method, it tests the method getPlayer by
102      *      creating mock players with wrong values and only initializing a specific
103      *      player then it would use getPlayer method to retrieve that specific
104      *      player and checks the length of the player's hand.
105      *
```

```
106      * @link Player.java
107      *
108      * @PlayerClassInstance player
109      * @PlayerClassMethods getPlayer(int)
110      */
111     @Test
112     public void testGetPlayer() {
113         try {
114             LinkedList<LinkedList<Integer>> players = new LinkedList<>();
115             for (int i = 0; i < 5; i++) {
116                 players.add(new LinkedList<>());
117             }
118             Player player = new Player(players);
119             Class<?> playerClass = player.getClass();
120             Field playersField = playerClass.getDeclaredField("players");
121             playersField.setAccessible(true);
122 
123             ((LinkedList<LinkedList<Integer>>) playersField.get(player)).get(3).add(null);
124             ((LinkedList<LinkedList<Integer>>) playersField.get(player)).get(3).add(null);
125             ((LinkedList<LinkedList<Integer>>) playersField.get(player)).get(3).add(null);
126             ((LinkedList<LinkedList<Integer>>) playersField.get(player)).get(3).add(null);
127 
128             assertEquals(4, player.getPlayer(3).size());
129         } catch (Exception e) {
130             fail("testGetPlayer Failed");
131         }
132     }
133 
134     /**
135      * @see testHasDuplicates
136      *
137      *      - testHasDuplicates is a void method, it tests the method hasDuplicates
138      *      by creating 3 mock hands with different problems with their size and
139      *      values
140      *      and pass each hand to this method, this method will return true if there
141      *      is a value repeated within the list even null values.
142      *
143      * @link Player.java
144      *
145      * @PlayerClassInstance player
146      * @PlayerClassMethods hasDuplicates(LinkedList<Integer>)
147      */
148     @Test
149     public void testHasDuplicates() {
150         Player player = new Player();
151 
152         LinkedList<Integer> hand = new LinkedList<>();
153         hand.add(0);
154         hand.add(-1);
155         hand.add(-1);
156         hand.add(-237);
157         hand.add(null);
158         hand.add(-237);
159         hand.add(-237);
```

```
160         hand.add(0);
161         hand.add(1);
162 
163         assertTrue(player.hasDuplicates(hand));
164 
165         hand = new LinkedList<>();
166         hand.add(null);
167         hand.add(-1);
168         hand.add(-0);
169         hand.add(1);
170 
171         assertTrue(!player.hasDuplicates(hand));
172 
173         hand = new LinkedList<>();
174         hand.add(null);
175         hand.add(null);
176         hand.add(null);
177         hand.add(null);
178 
179         assertTrue(player.hasDuplicates(hand));
180     }
181 
182     /**
183      * @see testGetCard
184      *
185      *      - testGetCard is a void method, this method creates an instance os the
186      *      player class, using that instance it would set the player number and set
187      *      players list with it and retrieves it with getPlayer method, then it
188      *      would create mock hands to test the robustness of the getCard method
189      *      which is our subject here.
190      *
191      * @Note getCard method returns 0 if there is a null value in the hand, it
192      *       returns the last element if there is no duplicate in the hand ot it
193      *       returns the
194      *       first occurrence of a digit which has not been repeated throughout the
195      *       hand.
196      *
197      * @link Player.java
198      *
199      * @PlayerClassInstance player
200      * @PlayerClassMethods getCard(LinkedList<Integer>), getPlayer(int),
201      *                     setPlayers(int)
202      *
203      */
204     @Test
205     public void testGetCard() {
206 
207         Player player = new Player();
208 
209         player.playerNumber = 1;
210         player.setPlayers(player.playerNumber);
211 
212         LinkedList<Integer> playerHand = player.getPlayer(0);
213         playerHand.add(9);
```

```
214         playerHand.add(-1);
215         playerHand.add(null);
216         playerHand.add(null);
217         playerHand.add(90);
218         playerHand.add(90);
219         playerHand.add(2);
220 
221         // return 0 if one or more elements in a hand are null (reporting a problem
222         // within the system)
223         assertEquals(0, player.getCard(playerHand));
224 
225         playerHand = new LinkedList<>();
226 
227         playerHand.add(2);
228         playerHand.add(9);
229         playerHand.add(-1);
230         playerHand.add(90);
231         playerHand.add(90);
232         playerHand.add(2);
233 
234         // return the first element that has no duplicate
235         assertEquals(9, player.getCard(playerHand));
236 
237         playerHand = new LinkedList<>();
238 
239         playerHand.add(-2);
240         playerHand.add(-1);
241         playerHand.add(0);
242         playerHand.add(1);
243         playerHand.add(2);
244 
245         // if not duplicates then return the last element
246         assertEquals(2, player.getCard(playerHand));
247 
248     }
249 
250     /**
251      * @see testSetPlayers
252      *
253      *      - testSetPlayers is a void method, it tests the setPlayers method by
254      *      passing it any wrong value possible and then by using getPlayers method
255      *      it checks the size of the created LinkedLists.
256      *
257      * @link Player.java
258      *
259      * @PlayerClassInstance player
260      * @PlayerClassMethods setPlayers(int), getPlayers()
261      */
262     @Test
263     public void testSetPlayers() {
264 
265         Player player = new Player();
266 
267         player.setPlayers(-1);
```

```
268         assertEquals(1, player.getPlayers().size());
269 
270         player.setPlayers(0);
271         assertEquals(1, Player.getPlayers().size());
272 
273         player.setPlayers(1);
274         assertEquals(1, player.getPlayers().size());
275 
276     }
277 }
278 
```

testInputOutput.java

```
1
2
3  import java.io.ByteArrayInputStream;
4  import java.io.File;
5  import java.io.InputStream;
6  import java.lang.reflect.Field;
7  import java.lang.reflect.Method;
8  import java.nio.file.Files;
9  import java.nio.file.Path;
10 import java.nio.file.Paths;
11 import java.util.LinkedList;
12 import org.junit.Test;
13 import static org.junit.Assert.*;
14
15 /**
16  * @see testInputOutput
17  *
18  *      - testInputOutput class is a test suit for the class with the
19  *      corresponding name (InputOutput Class),
20  *      It test the constructors, methods as well as Objects and instances
21  *      created and/or used by this method.
22  *
23  * @Note most test methods within this test class are testing multiple aspects
24  *       of the class such as other methods and objects
25  *
26  * @Note methods deleteDeckFiles(), deletePlayerFiles(), createDeckFiles(),
27  *       createPlayerFiles(), functionality has been tested from within other
28  *       test methods, these methods have only one dynamic value used within
29  *       them (playerNumber) which is being checked and validated inside each
30  *       method.
31  *
32  * @Note for some methods in this test class java reflection is used to access
33  *       private methods and instances of InputOutput class.
34  *
35  * @author Amirali Famili
36  */
37 public class testInputOutput {
38
39     /**
40      * @see testGetPlayerNumberValidInput
41      *
42      *      - testGetPlayerNumberValidInput is a void method, it uses the
43      *      InputStream to pass a mock input to the getPlayerNumber method and
44      *      compare the result with the mock valid input.
45      *
46      * @link InputOutput.java
47      *
48      * @Note since the method is working fine it's not asking for the player number
49      *       again recursively
50      *
51      * @InputOutputClassInstance user
```

```
52      * @InputOutputClassMethods getPlayerNumber()
53      */
54     @Test
55     public void testGetPlayerNumberValidInput() {
56         String input = "3\n";
57         InputStream in = new ByteArrayInputStream(input.getBytes());
58         System.setIn(in);
59 
60         InputOutput user = new InputOutput();
61         int result = user.getPlayerNumber();
62 
63         System.setIn(System.in);
64 
65         assertEquals(3, result);
66     }
67 
68     /**
69      * @see testGetPlayerNumberInvalidString
70      *
71      *      - testGetPlayerNumberInvalidInput is an unimplemented test method, it
72      *      uses input stream to pass a string to the getPlayerNumber method,
73      *      however since this method is recursively calling it self again until it
74      *      receives a valid Integer this method could not be implemented in the
75      *      test.
76      *
77      * @link InputOutput.java
78      *
79      * @Note note that this type of input can be checked directly from terminal when
80      *       the program asks for a player number as input
81      *
82      * @InputOutputClassInstance user
83      * @InputOutputClassMethods getPlayerNumber()
84      */
85     // @Test
86     public void testGetPlayerNumberInvalidString() {
87         String input = "invalid\n";
88         InputStream in = new ByteArrayInputStream(input.getBytes());
89         System.setIn(in);
90 
91         InputOutput user = new InputOutput();
92         try {
93             user.getPlayerNumber();
94             fail("Expected NumberFormatException");
95         } catch (NumberFormatException e) {
96         }
97         System.setIn(System.in);
98     }
99 
100    /**
101     * @see testGetPackFilePath
102     *
103     *      - testGetPackFilePath tests a valid mockPack which contains : 0,
104     *      positive and negative integers and checks the length of the list which
105     *      contains the extracted elements, the method getPackFilePath will ask for
```

```
106      *      a new pack via terminal if it detects any invalid element within the
107      *      pack file.
108      *
109      * @link InputOutput.java
110      *
111      * @Note any invalid pack file format like strings will be detected by the
112      *       program this can be checked via terminal, Negative Integers and Spaces
113      *       will be ignored
114      * @Note Integer overflow will not be tested here since the program will ask for
115      *       a "Valid" pack and the test will crash
116      *
117      * @InputOutputClassInstance file
118      * @InputOutputClassMethods getPackFilePath()
119      */
120     @Test
121     public void testGetPackFilePath() {
122         String input = "mockPack.txt\n";
123         InputStream in = new ByteArrayInputStream(input.getBytes());
124         System.setIn(in);
125
126         InputOutput file = new InputOutput();
127         LinkedList<Integer> pack = file.getPackFilePath();
128
129         assertEquals(29, pack.size());
130         System.setIn(System.in);
131     }
132
133     /**
134      * @see testInputOutputConstructorLinkedList
135      *
136      *      - testInputOutputConstructorLinkedList is a void method, responsible to
137      *      test InputOutput constructor with players nested LinkedList as argument,
138      *      this constructor will delete player and deck files and creates new ones
139      *      for the new players with the correct number of files, then it would add
140      *      the current hand of players to the player files to initialize the hand
141      *      they started the game with, this method will test all of them in order.
142      *
143      * @link InputOutput.java
144      *
145      * @Note the methods used within the constructor will be tested later in this
146      *       test class, this test class is for testing the functionality of the
147      *       constructor it self.
148      *
149      * @InputOutputClassInstance inputOutput, files
150      * @InputOutputClassMethods deleteDeckFiles(), deletePlayerFiles(),
151      */
152     @Test
153     public void testInputOutputConstructorLinkedList() {
154
155         try {
156             InputOutput inputOutput = new InputOutput();
157
158             Class<?> playerClass = inputOutput.getClass();
159             Method deleteDeckFilesMethod = playerClass.getDeclaredMethod("deleteDeckFiles");
```

```
160             Method deletePlayerFilesMethod = playerClass.getDeclaredMethod("deletePlayerFiles");
161
162             deleteDeckFilesMethod.setAccessible(true);
163             deletePlayerFilesMethod.setAccessible(true);
164
165             deleteDeckFilesMethod.invoke(inputOutput);
166             deletePlayerFilesMethod.invoke(inputOutput);
167
168             LinkedList<LinkedList<Integer>> players = new LinkedList<>();
169             for (int i = 0; i < 8; i++) {
170                 players.add(new LinkedList<>());
171                 for (int j = 0; j < 6; j++) {
172                     players.get(i).add(j);
173                 }
174             }
175
176             InputOutput files = new InputOutput(players);
177
178             File player_2 = new File("players/player" + 2 + "_output.txt");
179             File player_3 = new File("players/player" + 3 + "_output.txt");
180             File player_5 = new File("players/player" + 5 + "_output.txt");
181             File player_7 = new File("players/player" + 7 + "_output.txt");
182             File deck_1 = new File("decks/deck" + 1 + "_output.txt");
183             File deck_4 = new File("decks/deck" + 4 + "_output.txt");
184             File deck_6 = new File("decks/deck" + 6 + "_output.txt");
185             File deck_7 = new File("decks/deck" + 7 + "_output.txt");
186
187             assertTrue(player_2.exists());
188             assertTrue(player_3.exists());
189             assertTrue(player_5.exists());
190             assertTrue(player_7.exists());
191             assertTrue(deck_1.exists());
192             assertTrue(deck_4.exists());
193             assertTrue(deck_6.exists());
194             assertTrue(deck_7.exists());
195
196             Path playerPath = Paths.get("players/player7_output.txt");
197             Path deckPath = Paths.get("decks/deck7_output.txt");
198
199             assertTrue(Files.size(deckPath) == 0);
200             assertTrue(Files.size(playerPath) != 0);
201         } catch (Exception e) {
202             fail("testInputOutputConstructorLinkedList Failed");
203         }
204     }
205
206     /**
207      * @see testInputOutputConstructorLinkedLists
208      *
209      *      - testInputOutputConstructorLinkedLists is a void method, It's
210      *      responsible for testing the functionality of the InputOutput constructor
211      *      that takes winner, player and deck lists, constructor should write the
212      *      final decks in their file and inform all players that a player has won,
213      *      this method will check that it's actually doing those things.
```

```
214     *
215     * @link InputOutput.java
216     *
217     * @Note the methods used within the constructor will be tested later in this
218     *       test class, this test class is for testing the functionality of the
219     *       constructor it self.
220     *
221     * @InputOutputClassInstance end, files
222     * @InputOutputClassMethods deleteDeckFiles(), deletePlayerFiles(),
223     *                          createDeckFiles(), createPlayerFiles()
224     */
225    @Test
226    public void testInputOutputConstructorLinkedLists() {
227
228        try {
229            InputOutput inputOutput = new InputOutput();
230
231            Class<?> playerClass = inputOutput.getClass();
232            Method deleteDeckFilesMethod = playerClass.getDeclaredMethod("deleteDeckFiles");
233            Method deletePlayerFilesMethod = playerClass.getDeclaredMethod("deletePlayerFiles");
234            Method createDeckFilesMethod = playerClass.getDeclaredMethod("createDeckFiles");
235            Method createPlayerFilesMethod = playerClass.getDeclaredMethod("createPlayerFiles");
236            Field playerNumberField = playerClass.getDeclaredField("playerNumber");
237
238            deleteDeckFilesMethod.setAccessible(true);
239            deletePlayerFilesMethod.setAccessible(true);
240            createDeckFilesMethod.setAccessible(true);
241            createPlayerFilesMethod.setAccessible(true);
242            playerNumberField.setAccessible(true);
243
244            deleteDeckFilesMethod.invoke(inputOutput);
245            deletePlayerFilesMethod.invoke(inputOutput);
246            playerNumberField.set(inputOutput, 5);
247            createDeckFilesMethod.invoke(inputOutput);
248            createPlayerFilesMethod.invoke(inputOutput);
249
250            LinkedList<LinkedList<Integer>> players = new LinkedList<>();
251            LinkedList<LinkedList<Integer>> decks = new LinkedList<>();
252            for (int i = 0; i < 5; i++) {
253                players.add(new LinkedList<>());
254                decks.add(new LinkedList<>());
255                for (int j = 0; j < 6; j++) {
256                    players.get(i).add(j);
257                    decks.get(i).add(j);
258                }
259            }
260
261            LinkedList<Integer> winner = new LinkedList<>(players.get(4));
262
263            InputOutput end = new InputOutput(winner, players, decks);
264
265            Path playerPath = Paths.get("players/player5_output.txt");
266            Path deckPath = Paths.get("decks/deck5_output.txt");
267
```

```
268             assertTrue(Files.size(deckPath) != 0);
269             assertTrue(Files.size(playerPath) != 0);
270         } catch (Exception e) {
271             fail("testInputOutputConstructorLinkedLists Failed");
272         }
273     }
274 
275     /**
276      * @see testWriteEndGameSamePlayers
277      *
278      *     - testWriteEndGameSamePlayers is a void method, It tests the method
279      *      writeEndGame in the InputOutput Class by passing
280      *      the same player hands and same winner to it.
281      *
282      * @link InputOutput.java
283      *
284      * @Note one bug here is that player 5 has been declared as the winner however,
285      *       in the files player 1 is informing other players that he has won,
286      *       this is because the game should have one winner and it should be
287      *       impossible to have two winners with the same hand, therefore first
288      *       occurrence of "Winner"'s index is written in the files.
289      *
290      * @InputOutputClassInstance player
291      * @InputOutputClassMethods deleteDeckFiles(), deletePlayerFiles(),
292      *                          createDeckFiles(), createPlayerFiles(),
293      *                          writeEndGame(LinkedList<LinkedList<Integer>>,
294      *                          LinkedList<Integer>)
295      */
296     @Test
297     public void testWriteEndGameSamePlayers() {
298         try {
299             LinkedList<LinkedList<Integer>> players = new LinkedList<>();
300             for (int i = 0; i < 5; i++) {
301                 players.add(new LinkedList<>());
302                 for (int j = 0; j < 6; j++) {
303                     players.get(i).add(j);
304                 }
305             }
306 
307             InputOutput inputOutput = new InputOutput();
308 
309             Class<?> playerClass = inputOutput.getClass();
310             Method deleteDeckFilesMethod = playerClass.getDeclaredMethod("deleteDeckFiles");
311             Method deletePlayerFilesMethod = playerClass.getDeclaredMethod("deletePlayerFiles");
312             Method createDeckFilesMethod = playerClass.getDeclaredMethod("createDeckFiles");
313             Method createPlayerFilesMethod = playerClass.getDeclaredMethod("createPlayerFiles");
314             Method writeEndGameMethod = playerClass.getDeclaredMethod("writeEndGame", LinkedList.class,
315                     LinkedList.class);
316             Field playerNumberField = playerClass.getDeclaredField("playerNumber");
317 
318             deleteDeckFilesMethod.setAccessible(true);
319             deletePlayerFilesMethod.setAccessible(true);
320             createDeckFilesMethod.setAccessible(true);
321             createPlayerFilesMethod.setAccessible(true);
```

```
322             writeEndGameMethod.setAccessible(true);
323             playerNumberField.setAccessible(true);
324 
325             playerNumberField.set(inputOutput, 5);
326             deleteDeckFilesMethod.invoke(inputOutput);
327             deletePlayerFilesMethod.invoke(inputOutput);
328             createDeckFilesMethod.invoke(inputOutput);
329             createPlayerFilesMethod.invoke(inputOutput);
330             writeEndGameMethod.invoke(inputOutput, players, players.get(3));
331 
332             Path playerPath1 = Paths.get("players/player1_output.txt");
333             Path playerPath2 = Paths.get("players/player2_output.txt");
334             Path playerPath3 = Paths.get("players/player3_output.txt");
335             Path playerPath4 = Paths.get("players/player4_output.txt");
336             Path playerPath5 = Paths.get("players/player5_output.txt");
337 
338             assertTrue(Files.size(playerPath1) != 0);
339             assertTrue(Files.size(playerPath2) != 0);
340             assertTrue(Files.size(playerPath3) != 0);
341             assertTrue(Files.size(playerPath4) != 0);
342             assertTrue(Files.size(playerPath5) != 0);
343 
344         } catch (Exception e) {
345             fail("testWriteEndGameSamePlayers Failed");
346         }
347     }
348 
349     /**
350      * @see testWriteEndGameEmptyPlayers
351      *
352      *      - testWriteEndGameEmptyPlayers is a void method, It tests the method
353      *      writeEndGame in the InputOutput Class by passing
354      *      empty players and winner.
355      *
356      * @link InputOutput.java
357      *
358      * @InputOutputClassInstance player
359      * @InputOutputClassMethods deleteDeckFiles(), deletePlayerFiles(),
360      *                          createDeckFiles(), createPlayerFiles(),
361      *                          writeEndGame(LinkedList<LinkedList<Integer>>,
362      *                          LinkedList<Integer>)
363      */
364     @Test
365     public void testWriteEndGameEmptyPlayers() {
366         try {
367             LinkedList<LinkedList<Integer>> players = new LinkedList<>();
368             for (int i = 0; i < 5; i++) {
369                 players.add(new LinkedList<>());
370             }
371 
372             InputOutput inputOutput = new InputOutput();
373 
374             Class<?> playerClass = inputOutput.getClass();
375             Method deleteDeckFilesMethod = playerClass.getDeclaredMethod("deleteDeckFiles");
```

```
376             Method deletePlayerFilesMethod = playerClass.getDeclaredMethod("deletePlayerFiles");
377             Method createDeckFilesMethod = playerClass.getDeclaredMethod("createDeckFiles");
378             Method createPlayerFilesMethod = playerClass.getDeclaredMethod("createPlayerFiles");
379             Method writeEndGameMethod = playerClass.getDeclaredMethod("writeEndGame", LinkedList.class,
380                     LinkedList.class);
381             Field playerNumberField = playerClass.getDeclaredField("playerNumber");
382 
383             deleteDeckFilesMethod.setAccessible(true);
384             deletePlayerFilesMethod.setAccessible(true);
385             createDeckFilesMethod.setAccessible(true);
386             createPlayerFilesMethod.setAccessible(true);
387             writeEndGameMethod.setAccessible(true);
388             playerNumberField.setAccessible(true);
389 
390             playerNumberField.set(inputOutput, 5);
391             deleteDeckFilesMethod.invoke(inputOutput);
392             deletePlayerFilesMethod.invoke(inputOutput);
393             createDeckFilesMethod.invoke(inputOutput);
394             createPlayerFilesMethod.invoke(inputOutput);
395             writeEndGameMethod.invoke(inputOutput, players, players.get(3));
396 
397             Path playerPath1 = Paths.get("players/player1_output.txt");
398             Path playerPath2 = Paths.get("players/player2_output.txt");
399             Path playerPath3 = Paths.get("players/player3_output.txt");
400             Path playerPath4 = Paths.get("players/player4_output.txt");
401             Path playerPath5 = Paths.get("players/player5_output.txt");
402 
403             assertTrue(Files.size(playerPath1) != 0);
404             assertTrue(Files.size(playerPath2) != 0);
405             assertTrue(Files.size(playerPath3) != 0);
406             assertTrue(Files.size(playerPath4) != 0);
407             assertTrue(Files.size(playerPath5) != 0);
408 
409         } catch (Exception e) {
410             fail("testWriteEndGameEmptyPlayers Failed");
411         }
412     }
413 
414     /**
415      * @see testWriteEndGameNoWinner
416      *
417      *      - testWriteEndGameNoWinner is a void method, It tests the method
418      *      writeEndGame in the InputOutput Class by passing
419      *      an empty winner hand which is not present in the original valid players
420      *      list.
421      *
422      * @link InputOutput.java
423      *
424      * @Note the method will successfully write the correct format information in
425      *       the text files however, since we don't have a winner, the winner is
426      *       declared as player 0.
427      *
428      * @InputOutputClassInstance player
429      * @InputOutputClassMethods deleteDeckFiles(), deletePlayerFiles(),
```

```
430     *                   createDeckFiles(), createPlayerFiles(),
431     *                   writeEndGame(LinkedList<LinkedList<Integer>>,
432     *                   LinkedList<Integer>)
433     */
434    @Test
435    public void testWriteEndGameNoWinner() {
436       try {
437          LinkedList<LinkedList<Integer>> players = new LinkedList<>();
438          for (int i = 0; i < 5; i++) {
439             players.add(new LinkedList<>());
440             for (int j = 0; j < 6; j++) {
441                players.get(i).add(j);
442             }
443          }
444
445          players.add(null); // add a null value instead of hand
446
447          InputOutput inputOutput = new InputOutput();
448
449          Class<?> playerClass = inputOutput.getClass();
450          Method deleteDeckFilesMethod = playerClass.getDeclaredMethod("deleteDeckFiles");
451          Method deletePlayerFilesMethod = playerClass.getDeclaredMethod("deletePlayerFiles");
452          Method createDeckFilesMethod = playerClass.getDeclaredMethod("createDeckFiles");
453          Method createPlayerFilesMethod = playerClass.getDeclaredMethod("createPlayerFiles");
454          Method writeEndGameMethod = playerClass.getDeclaredMethod("writeEndGame", LinkedList.class,
455                LinkedList.class);
456          Field playerNumberField = playerClass.getDeclaredField("playerNumber");
457
458          deleteDeckFilesMethod.setAccessible(true);
459          deletePlayerFilesMethod.setAccessible(true);
460          createDeckFilesMethod.setAccessible(true);
461          createPlayerFilesMethod.setAccessible(true);
462          writeEndGameMethod.setAccessible(true);
463          playerNumberField.setAccessible(true);
464
465          playerNumberField.set(inputOutput, 6);
466          deleteDeckFilesMethod.invoke(inputOutput);
467          deletePlayerFilesMethod.invoke(inputOutput);
468          createDeckFilesMethod.invoke(inputOutput);
469          createPlayerFilesMethod.invoke(inputOutput);
470          writeEndGameMethod.invoke(inputOutput, players, players.get(5));
471
472          Path playerPath1 = Paths.get("players/player1_output.txt");
473          Path playerPath2 = Paths.get("players/player2_output.txt");
474          Path playerPath3 = Paths.get("players/player3_output.txt");
475          Path playerPath4 = Paths.get("players/player4_output.txt");
476          Path playerPath5 = Paths.get("players/player5_output.txt");
477          Path playerPath6 = Paths.get("players/player6_output.txt");
478
479          assertTrue(Files.size(playerPath1) != 0);
480          assertTrue(Files.size(playerPath2) != 0);
481          assertTrue(Files.size(playerPath3) != 0);
482          assertTrue(Files.size(playerPath4) != 0);
```

```
483             assertTrue(Files.size(playerPath5) != 0);
484             assertTrue(Files.size(playerPath6) == 0); // nothing should be written since the player is null
485 
486         } catch (Exception e) {
487             fail("testWriteEndGameNoWinner Failed");
488         }
489     }
490 
491     /**
492      * @see testWriteEndGameNull
493      *
494      *      - testWriteEndGameNull is a void method, It tests the method
495      *      writeEndGame in the InputOutput Class by passing
496      *      null values instead of players and winner.
497      *
498      * @link InputOutput.java
499      *
500      * @Note the method shouldn't print anything to the files but it should execute
501      *       the code with no errors.
502      *
503      * @InputOutputClassInstance player
504      * @InputOutputClassMethods deleteDeckFiles(), deletePlayerFiles(),
505      *                          createDeckFiles(), createPlayerFiles(),
506      *                          writeEndGame(LinkedList<LinkedList<Integer>>,
507      *                          LinkedList<Integer>)
508      */
509     @Test
510     public void testWriteEndGameNull() {
511         try {
512             InputOutput inputOutput = new InputOutput();
513 
514             Class<?> playerClass = inputOutput.getClass();
515             Method deleteDeckFilesMethod = playerClass.getDeclaredMethod("deleteDeckFiles");
516             Method deletePlayerFilesMethod = playerClass.getDeclaredMethod("deletePlayerFiles");
517             Method createDeckFilesMethod = playerClass.getDeclaredMethod("createDeckFiles");
518             Method createPlayerFilesMethod = playerClass.getDeclaredMethod("createPlayerFiles");
519             Method writeEndGameMethod = playerClass.getDeclaredMethod("writeEndGame", LinkedList.class,
520                     LinkedList.class);
521             Field playerNumberField = playerClass.getDeclaredField("playerNumber");
522 
523             deleteDeckFilesMethod.setAccessible(true);
524             deletePlayerFilesMethod.setAccessible(true);
525             createDeckFilesMethod.setAccessible(true);
526             createPlayerFilesMethod.setAccessible(true);
527             writeEndGameMethod.setAccessible(true);
528             playerNumberField.setAccessible(true);
529 
530             playerNumberField.set(inputOutput, 6);
531             deleteDeckFilesMethod.invoke(inputOutput);
532             deletePlayerFilesMethod.invoke(inputOutput);
533             createDeckFilesMethod.invoke(inputOutput);
534             createPlayerFilesMethod.invoke(inputOutput);
535             writeEndGameMethod.invoke(inputOutput, null, null);
536 
```

```
537             Path playerPath1 = Paths.get("players/player1_output.txt");
538             Path playerPath2 = Paths.get("players/player2_output.txt");
539             Path playerPath3 = Paths.get("players/player3_output.txt");
540             Path playerPath4 = Paths.get("players/player4_output.txt");
541             Path playerPath5 = Paths.get("players/player5_output.txt");
542
543             assertTrue(Files.size(playerPath1) == 0);
544             assertTrue(Files.size(playerPath2) == 0);
545             assertTrue(Files.size(playerPath3) == 0);
546             assertTrue(Files.size(playerPath4) == 0);
547             assertTrue(Files.size(playerPath5) == 0);
548
549         } catch (Exception e) {
550             fail("testWriteEndGameNull Failed");
551         }
552     }
553
554     /**
555      * @see testWriteDrawsCard
556      *
557      *      - testWriteDrawsCard is a void method, It tests the method
558      *      writeDrawsCard in the InputOutput Class by passing wrong values as
559      *      player index and check to see if something will be written in their file
560      *      or an error will occur.
561      *
562      * @link InputOutput.java
563      *
564      * @Note since the drawnCard argument inside writeDrawsCard method doesn't need
565      *       any validation checks, since it's just writing the argument as a
566      *       string, I have avoided testing this parameter.
567      *
568      * @InputOutputClassInstance draw
569      * @InputOutputClassMethods deleteDeckFiles(), deletePlayerFiles(),
570      *                          createDeckFiles(), createPlayerFiles(),
571      *                          writeDrawsCard(int, int)
572      */
573     @Test
574     public void testWriteDrawsCard() {
575         try {
576             InputOutput inputOutput = new InputOutput();
577
578             Class<?> playerClass = inputOutput.getClass();
579             Method deleteDeckFilesMethod = playerClass.getDeclaredMethod("deleteDeckFiles");
580             Method deletePlayerFilesMethod = playerClass.getDeclaredMethod("deletePlayerFiles");
581             Method createDeckFilesMethod = playerClass.getDeclaredMethod("createDeckFiles");
582             Method writeDrawsCardMethod = playerClass.getDeclaredMethod("writeDrawsCard", int.class, int.class);
583             Method createPlayerFilesMethod = playerClass.getDeclaredMethod("createPlayerFiles");
584             Field playerNumberField = playerClass.getDeclaredField("playerNumber");
585
586             deleteDeckFilesMethod.setAccessible(true);
587             deletePlayerFilesMethod.setAccessible(true);
588             createDeckFilesMethod.setAccessible(true);
589             createPlayerFilesMethod.setAccessible(true);
590             playerNumberField.setAccessible(true);
```

```
591             writeDrawsCardMethod.setAccessible(true);
592
593             deleteDeckFilesMethod.invoke(inputOutput);
594             deletePlayerFilesMethod.invoke(inputOutput);
595             playerNumberField.set(inputOutput, 1);
596             createDeckFilesMethod.invoke(inputOutput);
597             createPlayerFilesMethod.invoke(inputOutput);
598
599             writeDrawsCardMethod.invoke(inputOutput, -8743, -3);
600
601             Path playerPath1 = Paths.get("players/player1_output.txt");
602
603             // the file should be empty because a player with index -3 does not exists
604             assertTrue(Files.size(playerPath1) == 0);
605
606             writeDrawsCardMethod.invoke(inputOutput, -8743, 1);
607             // the file shouldn't be empty since the player has drawn a -8374 from their
608             // left deck
609             assertTrue(Files.size(playerPath1) != 0);
610
611         } catch (Exception e) {
612             fail("testWriteDrawsCard Failed");
613         }
614     }
615
616     /**
617      * @see testWriteDiscardsCard
618      *
619      *      - testWriteDiscardsCard is a void method, It tests the method
620      *      writeDiscardsCard in the InputOutput Class by passing wrong values as
621      *      player index and check to see if something will be written in their file
622      *      or an error will occur.
623      *
624      * @link InputOutput.java
625      *
626      * @Note since the discardedCard argument inside writeDiscardsCard method
627      *       doesn't need
628      *       any validation checks, since it's just writing the argument as a
629      *       string, I have avoided testing this parameter.
630      *
631      * @InputOutputClassInstance discard
632      * @InputOutputClassMethods deleteDeckFiles(), deletePlayerFiles(),
633      *                          createDeckFiles(), createPlayerFiles(),
634      *                          writeDiscardsCard(int, int)
635      */
636     @Test
637     public void testWriteDiscardsCard() {
638         try {
639             InputOutput inputOutput = new InputOutput();
640
641             Class<?> playerClass = inputOutput.getClass();
642             Method deleteDeckFilesMethod = playerClass.getDeclaredMethod("deleteDeckFiles");
643             Method deletePlayerFilesMethod = playerClass.getDeclaredMethod("deletePlayerFiles");
644             Method createDeckFilesMethod = playerClass.getDeclaredMethod("createDeckFiles");
```

```
645             Method writeDiscardsCardMethod = playerClass.getDeclaredMethod("writeDiscardsCard", int.class, int.class);
646             Method createPlayerFilesMethod = playerClass.getDeclaredMethod("createPlayerFiles");
647             Field playerNumberField = playerClass.getDeclaredField("playerNumber");
648 
649             deleteDeckFilesMethod.setAccessible(true);
650             deletePlayerFilesMethod.setAccessible(true);
651             createDeckFilesMethod.setAccessible(true);
652             createPlayerFilesMethod.setAccessible(true);
653             playerNumberField.setAccessible(true);
654             writeDiscardsCardMethod.setAccessible(true);
655 
656             deleteDeckFilesMethod.invoke(inputOutput);
657             deletePlayerFilesMethod.invoke(inputOutput);
658             playerNumberField.set(inputOutput, 3);
659             createDeckFilesMethod.invoke(inputOutput);
660             createPlayerFilesMethod.invoke(inputOutput);
661 
662             writeDiscardsCardMethod.invoke(inputOutput, -843, -37);
663             writeDiscardsCardMethod.invoke(inputOutput, -843, -38);
664             writeDiscardsCardMethod.invoke(inputOutput, -843, -3934);
665 
666             Path playerPath1 = Paths.get("players/player1_output.txt");
667             Path playerPath2 = Paths.get("players/player2_output.txt");
668             Path playerPath3 = Paths.get("players/player3_output.txt");
669 
670             assertTrue(Files.size(playerPath1) == 0);
671             assertTrue(Files.size(playerPath2) == 0);
672             assertTrue(Files.size(playerPath3) == 0);
673 
674             writeDiscardsCardMethod.invoke(inputOutput, -843, 1);
675             writeDiscardsCardMethod.invoke(inputOutput, -843, 2);
676             writeDiscardsCardMethod.invoke(inputOutput, -843, 3);
677 
678             assertTrue(Files.size(playerPath1) != 0);
679             assertTrue(Files.size(playerPath2) != 0);
680             assertTrue(Files.size(playerPath3) != 0);
681 
682         } catch (Exception e) {
683             fail("testWriteDiscardsCard Failed");
684         }
685     }
686 
687     /**
688      * @see testWriteCurrentHand
689      *
690      *      - testWriteCurrentHand is a void method, It tests the method
691      *      writeCurrentHand in the InputOutput Class by passing wrong values as
692      *      player index and player's hand to check if something will be written in
693      *      their file
694      *      or an error will occur.
695      *
696      * @Note this method will check the size of the file created to validate that
697      *       the correct information is being printed in the files, it's also
698      *       checking if the current hand is being added successfully.
```

```
699      *
700      * @link InputOutput.java
701      *
702      * @InputOutputClassInstance current
703      * @InputOutputClassMethods deleteDeckFiles(), deletePlayerFiles(),
704      *                          createDeckFiles(), createPlayerFiles(),
705      *                          writeCurrentHand(LinkedList<Integer>, int)
706      */
707     @Test
708     public void testWriteCurrentHand() {
709         try {
710             InputOutput inputOutput = new InputOutput();
711
712             Class<?> playerClass = inputOutput.getClass();
713             Method deleteDeckFilesMethod = playerClass.getDeclaredMethod("deleteDeckFiles");
714             Method deletePlayerFilesMethod = playerClass.getDeclaredMethod("deletePlayerFiles");
715             Method createDeckFilesMethod = playerClass.getDeclaredMethod("createDeckFiles");
716             Method writeCurrentHandMethod = playerClass.getDeclaredMethod("writeCurrentHand", LinkedList.class,
717                     int.class);
718             Method createPlayerFilesMethod = playerClass.getDeclaredMethod("createPlayerFiles");
719             Field playerNumberField = playerClass.getDeclaredField("playerNumber");
720
721             deleteDeckFilesMethod.setAccessible(true);
722             deletePlayerFilesMethod.setAccessible(true);
723             createDeckFilesMethod.setAccessible(true);
724             createPlayerFilesMethod.setAccessible(true);
725             playerNumberField.setAccessible(true);
726             writeCurrentHandMethod.setAccessible(true);
727
728             deleteDeckFilesMethod.invoke(inputOutput);
729             deletePlayerFilesMethod.invoke(inputOutput);
730             playerNumberField.set(inputOutput, 2);
731             createDeckFilesMethod.invoke(inputOutput);
732             createPlayerFilesMethod.invoke(inputOutput);
733
734             Path playerPath1 = Paths.get("players/player1_output.txt");
735             Path playerPath2 = Paths.get("players/player2_output.txt");
736
737             // player is null
738             writeCurrentHandMethod.invoke(inputOutput, null, 1);
739             writeCurrentHandMethod.invoke(inputOutput, null, 2);
740
741             assertTrue(Files.size(playerPath1) == 0);
742             assertTrue(Files.size(playerPath2) == 0);
743
744             // indexes are wrong
745             writeCurrentHandMethod.invoke(inputOutput, new LinkedList<>(), 0);
746             writeCurrentHandMethod.invoke(inputOutput, new LinkedList<>(), -3);
747
748             assertTrue(Files.size(playerPath1) == 0);
749             assertTrue(Files.size(playerPath2) == 0);
750
751             // correct
752             writeCurrentHandMethod.invoke(inputOutput, new LinkedList<>(), 1);
```

```
753             writeCurrentHandMethod.invoke(inputOutput, new LinkedList<>(), 2);
754
755             assertEquals(26, Files.size(playerPath1));
756             assertEquals(26, Files.size(playerPath2));
757
758             LinkedList<Integer> mockHand = new LinkedList<>();
759             mockHand.add(null);
760             mockHand.add(-2);
761             mockHand.add(-1);
762             mockHand.add(0);
763             mockHand.add(1);
764             mockHand.add(2);
765
766             writeCurrentHandMethod.invoke(inputOutput, mockHand, 1);
767             writeCurrentHandMethod.invoke(inputOutput, mockHand, 2);
768
769             assertEquals(65, Files.size(playerPath1));
770             assertEquals(65, Files.size(playerPath2));
771
772         } catch (Exception e) {
773             fail("testWriteCurrentHand Failed");
774         }
775     }
776
777     /**
778      * @see testWriteDeckContents
779      *
780      *      - testWriteDeckContents is a void method, It tests the method
781      *      writeDeckContents in the InputOutput Class by passing wrong values as
782      *      decks LinkedList and see if we get an error, it also validates that the
783      *      correct information is being written in the text files.
784      *
785      * @Note this method will check the size of the file created to validate that
786      *       the correct information is being printed in the files.
787      *
788      * @link InputOutput.java
789      *
790      * @InputOutputClassInstance deck
791      * @InputOutputClassMethods deleteDeckFiles(), deletePlayerFiles(),
792      *                          createDeckFiles(), createPlayerFiles(),
793      *                          writeDeckContents(LinkedList<LinkedList<Integer>>)
794      */
795     @Test
796     public void testWriteDeckContents() {
797
798         try {
799             InputOutput inputOutput = new InputOutput();
800
801             Class<?> playerClass = inputOutput.getClass();
802             Method deleteDeckFilesMethod = playerClass.getDeclaredMethod("deleteDeckFiles");
803             Method deletePlayerFilesMethod = playerClass.getDeclaredMethod("deletePlayerFiles");
804             Method createDeckFilesMethod = playerClass.getDeclaredMethod("createDeckFiles");
805             Method writeDeckContentsMethod = playerClass.getDeclaredMethod("writeDeckContents", LinkedList.class);
```

```
806             Field playerNumberField = playerClass.getDeclaredField("playerNumber");
807
808             deleteDeckFilesMethod.setAccessible(true);
809             deletePlayerFilesMethod.setAccessible(true);
810             createDeckFilesMethod.setAccessible(true);
811             playerNumberField.setAccessible(true);
812             writeDeckContentsMethod.setAccessible(true);
813
814             deleteDeckFilesMethod.invoke(inputOutput);
815             deletePlayerFilesMethod.invoke(inputOutput);
816             playerNumberField.set(inputOutput, 3);
817             createDeckFilesMethod.invoke(inputOutput);
818
819             Path deckPath1 = Paths.get("decks/deck1_output.txt");
820             Path deckPath2 = Paths.get("decks/deck2_output.txt");
821             Path deckPath3 = Paths.get("decks/deck3_output.txt");
822             Path deckPath4 = Paths.get("decks/deck4_output.txt");
823
824             writeDeckContentsMethod.invoke(inputOutput, new LinkedList<>());
825
826             assertEquals(0, Files.size(deckPath1));
827             assertEquals(0, Files.size(deckPath2));
828             assertEquals(0, Files.size(deckPath3));
829
830             LinkedList<LinkedList<Integer>> decks = new LinkedList<>();
831             for (int i = 0; i < 3; i++) {
832                 decks.add(new LinkedList<>());
833             }
834
835             decks.get(0).add(null);
836             decks.get(0).add(0);
837             decks.get(0).add(1);
838
839             decks.get(1).add(null);
840             decks.get(1).add(-1);
841
842             decks.get(2).add(null);
843             decks.get(2).add(34);
844             decks.get(2).add(null);
845             decks.get(2).add(-15);
846
847             writeDeckContentsMethod.invoke(inputOutput, decks);
848
849             assertEquals(22, Files.size(deckPath1));
850             assertEquals(21, Files.size(deckPath2));
851             assertEquals(26, Files.size(deckPath3));
852
853         } catch (Exception e) {
854             fail("testWriteDeckContents Failed");
855         }
856     }
857
858     /**
859      * @see testCardsToString
```

```
860      *
861      *      - testCardsToString is a void method, it tests the method cardsToString
862      *      method by creating a mock hand with wrong format and values and passing
863      *      it to this method without any errors.
864      *
865      * @link InputOutput.java
866      *
867      * @InputOutputClassInstance sHand
868      * @InputOutputClassMethods cardsToString(LinkedList<Integer>)
869      */
870     @Test
871     public void testCardsToString() {
872
873         InputOutput sHand = new InputOutput();
874
875         LinkedList<Integer> hand = new LinkedList<>();
876         hand.add(0);
877         hand.add(-1);
878         hand.add(-1);
879         hand.add(-237);
880         hand.add(null);
881         hand.add(-237);
882         hand.add(-237);
883         hand.add(0);
884         hand.add(1);
885
886         try {
887             Class<?> inputOutputClass = sHand.getClass();
888             Method cardsToStringMethod = inputOutputClass.getDeclaredMethod("cardsToString", LinkedList.class);
889             cardsToStringMethod.setAccessible(true);
890
891             String actual = (String) cardsToStringMethod.invoke(sHand, hand);
892             assertEquals(28, actual.length());
893             // since there are 19 characters and 9 spaces , given that null is counted as a
894             // space
895
896             // assertEquals("", (String) cardsToStringMethod.invoke(sHand, null)); // null
897             // values are not supported by java reflection but the code is checking for null
898             assertEquals("", (String) cardsToStringMethod.invoke(sHand, new LinkedList<>()));
899         } catch (Exception e) {
900             fail("testCardsToString Failed");
901         }
902     }
903
904     /**
905      * @see testInitialHand
906      *
907      *      - testInitialHand is a void method, it tests the method initialHands
908      *      from InputOutput class, initialHands method writes the very first lines
909      *      of text inside the players text files, this method is testing it's
910      *      functionality and robustness.
911      *
912      * @link InputOutput.java
913      *
```

```
914      * @InputOutputClassInstance player
915      * @InputOutputClassMethods initialHands(LinkedList<LinkedList<Integer>>)
916      */
917     @Test
918     public void testInitialHand() {
919 
920         try {
921             InputOutput inputOutput = new InputOutput();
922 
923             Class<?> playerClass = inputOutput.getClass();
924             Method deleteDeckFilesMethod = playerClass.getDeclaredMethod("deleteDeckFiles");
925             Method deletePlayerFilesMethod = playerClass.getDeclaredMethod("deletePlayerFiles");
926             Method createDeckFilesMethod = playerClass.getDeclaredMethod("createDeckFiles");
927             Method initialHandMethod = playerClass.getDeclaredMethod("initialHand", LinkedList.class);
928             Method createPlayerFilesMethod = playerClass.getDeclaredMethod("createPlayerFiles");
929             Field playerNumberField = playerClass.getDeclaredField("playerNumber");
930 
931             deleteDeckFilesMethod.setAccessible(true);
932             deletePlayerFilesMethod.setAccessible(true);
933             createDeckFilesMethod.setAccessible(true);
934             createPlayerFilesMethod.setAccessible(true);
935             playerNumberField.setAccessible(true);
936             initialHandMethod.setAccessible(true);
937 
938             deleteDeckFilesMethod.invoke(inputOutput);
939             deletePlayerFilesMethod.invoke(inputOutput);
940             playerNumberField.set(inputOutput, 4);
941             createDeckFilesMethod.invoke(inputOutput);
942             createPlayerFilesMethod.invoke(inputOutput);
943 
944             Path playerPath1 = Paths.get("players/player1_output.txt");
945             Path playerPath2 = Paths.get("players/player2_output.txt");
946             Path playerPath3 = Paths.get("players/player3_output.txt");
947             Path playerPath4 = Paths.get("players/player4_output.txt");
948 
949             LinkedList<LinkedList<Integer>> players = new LinkedList<>();
950             for (int i = 0; i < 3; i++) {
951                 players.add(new LinkedList<>());
952             }
953 
954             players.add(null);
955 
956             players.get(0).add(null);
957             players.get(0).add(null);
958             players.get(0).add(null);
959             players.get(0).add(null);
960             players.get(1).add(5);
961             players.get(1).add(-45);
962             players.get(1).add(-7834);
963             players.get(2).add(1);
964             players.get(2).add(1);
965             players.get(2).add(345678976);
966 
967             initialHandMethod.invoke(inputOutput, players);
```

```
968
969             assertEquals(27, Files.size(playerPath1));
970             assertEquals(35, Files.size(playerPath2));
971             assertEquals(37, Files.size(playerPath3));
972             assertEquals(0, Files.size(playerPath4));
973
974         } catch (Exception e) {
975             fail("testWriteDrawsCard Failed");
976         }
977     }
978     public static void main(String[] args) {
979         testInputOutput test = new testInputOutput();
980         test.testWriteDiscardsCard();
981     }
982 }
983
```

testRunner.java

```
1
2
3  import org.junit.runner.JUnitCore;
4  import org.junit.runner.Result;
5  import org.junit.runner.notification.Failure;
6
7
8  /**
9   * @see testRunner
10  *
11  *      - Class testRunner is the main class for the tests, it would run all the tests at the specified order, then it would print
    the final result at the terminal.
12  *
13  * @Note result.wasSuccessful() will return true if all the test have passed and false if even on hasn't so in the terminal :
14  *
15  *      All tests passed: true -> means that tests were successful
16  *      All tests passed: false -> means that tests were not successful
17  *
18  * @Note by running the main method of this class you can check the result of all the tests.
19  *
20  * @author Amirali Famili
21  */
22 public class testRunner {
23     public static void main(String[] args) {
24         Result result = JUnitCore.runClasses(testCardGame.class, testPlayer.class, testCard.class,
25                 testInputOutput.class);
26         for (Failure failure : result.getFailures()) {
27             System.out.println(failure.getMessage());
28         }
29
30         System.out.println("All tests passed: " + result.wasSuccessful());
31
32         System.exit(0);
33     }
34 }
```