

PMR: This program was really easy since they provided majority of the code. The only tricky part was making the testers work correctly. I also like how to provided a screen of the output since there are two tester files and I wasn’t sure which one would contain the correct input.

/\*\*

\* Card.java

\*

\* <code>Card</code> represents a playing card.

\*/

public class Card {

/\*\*

\* String value that holds the suit of the card

\*/

private String suit;

/\*\*

\* String value that holds the rank of the card

\*/

private String rank;

/\*\*

\* int value that holds the point value.

\*/

private int pointValue;

/\*\*

\* Creates a new <code>Card</code> instance.

\*

\* @param cardRank a <code>String</code> value

\* containing the rank of the card

\* @param cardSuit a <code>String</code> value

\* containing the suit of the card

\* @param cardPointValue an <code>int</code> value

\* containing the point value of the card

\*/

public Card(String cardRank, String cardSuit, int cardPointValue)

{

rank = cardRank;

suit = cardSuit;

pointValue = cardPointValue;

}

/\*\*

\* Accesses this <code>Card's</code> suit.

\* @return this <code>Card's</code> suit.

\*/

public String suit()

{

return suit;

}

/\*\*

\* Accesses this <code>Card's</code> rank.

\* @return this <code>Card's</code> rank.

\*/

public String rank()

{

return rank;

}

/\*\*

\* Accesses this <code>Card's</code> point value.

\* @return this <code>Card's</code> point value.

\*/

public int pointValue()

{

return pointValue;

}

/\*\* Compare this card with the argument.

\* @param otherCard the other card to compare to this

\* @return true if the rank, suit, and point value of this card

\* are equal to those of the argument;

\* false otherwise.

\*/

public boolean matches(Card otherCard)

{

if(rank.compareTo(otherCard.rank()) == 0 && suit.compareTo(otherCard.suit()) == 0 && pointValue == otherCard.pointValue())

{

return true;

}

else

{

return false;

}

}

/\*\*

\* Converts the rank, suit, and point value into a string in the format

\* "[Rank] of [Suit] (point value = [PointValue])".

\* This provides a useful way of printing the contents

\* of a <code>Deck</code> in an easily readable format or performing

\* other similar functions.

\*

\* @return a <code>String</code> containing the rank, suit,

\* and point value of the card.

\*/

@Override

public String toString()

{

String printme = rank + " of " + suit + " (point value = " + pointValue + ")";

return printme;

}

}

import java.awt.Point;

import java.awt.Graphics;

import java.awt.Dimension;

import java.awt.Font;

import java.awt.Color;

import java.awt.Toolkit;

import java.awt.event.ActionListener;

import java.awt.event.ActionEvent;

import java.awt.event.MouseListener;

import java.awt.event.MouseEvent;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.JButton;

import javax.swing.JLabel;

import javax.swing.ImageIcon;

import java.net.URL;

import java.util.List;

import java.util.ArrayList;

/\*\*

\* This class provides a GUI for solitaire games related to Elevens.

\*/

public class CardGameGUI extends JFrame implements ActionListener {

/\*\* Height of the game frame. \*/

private static final int DEFAULT\_HEIGHT = 302;

/\*\* Width of the game frame. \*/

private static final int DEFAULT\_WIDTH = 800;

/\*\* Width of a card. \*/

private static final int CARD\_WIDTH = 73;

/\*\* Height of a card. \*/

private static final int CARD\_HEIGHT = 97;

/\*\* Row (y coord) of the upper left corner of the first card. \*/

private static final int LAYOUT\_TOP = 30;

/\*\* Column (x coord) of the upper left corner of the first card. \*/

private static final int LAYOUT\_LEFT = 30;

/\*\* Distance between the upper left x coords of

\* two horizonally adjacent cards. \*/

private static final int LAYOUT\_WIDTH\_INC = 100;

/\*\* Distance between the upper left y coords of

\* two vertically adjacent cards. \*/

private static final int LAYOUT\_HEIGHT\_INC = 125;

/\*\* y coord of the "Replace" button. \*/

private static final int BUTTON\_TOP = 30;

/\*\* x coord of the "Replace" button. \*/

private static final int BUTTON\_LEFT = 570;

/\*\* Distance between the tops of the "Replace" and "Restart" buttons. \*/

private static final int BUTTON\_HEIGHT\_INC = 50;

/\*\* y coord of the "n undealt cards remain" label. \*/

private static final int LABEL\_TOP = 160;

/\*\* x coord of the "n undealt cards remain" label. \*/

private static final int LABEL\_LEFT = 540;

/\*\* Distance between the tops of the "n undealt cards" and

\* the "You lose/win" labels. \*/

private static final int LABEL\_HEIGHT\_INC = 35;

/\*\* The board (Board subclass). \*/

private Board board;

/\*\* The main panel containing the game components. \*/

private JPanel panel;

/\*\* The Replace button. \*/

private JButton replaceButton;

/\*\* The Restart button. \*/

private JButton restartButton;

/\*\* The "number of undealt cards remain" message. \*/

private JLabel statusMsg;

/\*\* The "you've won n out of m games" message. \*/

private JLabel totalsMsg;

/\*\* The card displays. \*/

private JLabel[] displayCards;

/\*\* The win message. \*/

private JLabel winMsg;

/\*\* The loss message. \*/

private JLabel lossMsg;

/\*\* The coordinates of the card displays. \*/

private Point[] cardCoords;

/\*\* kth element is true iff the user has selected card #k. \*/

private boolean[] selections;

/\*\* The number of games won. \*/

private int totalWins;

/\*\* The number of games played. \*/

private int totalGames;

/\*\*

\* Initialize the GUI.

\* @param gameBoard is a <code>Board</code> subclass.

\*/

public CardGameGUI(Board gameBoard) {

board = gameBoard;

totalWins = 0;

totalGames = 0;

// Initialize cardCoords using 5 cards per row

cardCoords = new Point[board.size()];

int x = LAYOUT\_LEFT;

int y = LAYOUT\_TOP;

for (int i = 0; i < cardCoords.length; i++) {

cardCoords[i] = new Point(x, y);

if (i % 5 == 4) {

x = LAYOUT\_LEFT;

y += LAYOUT\_HEIGHT\_INC;

} else {

x += LAYOUT\_WIDTH\_INC;

}

}

selections = new boolean[board.size()];

initDisplay();

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

repaint();

}

/\*\*

\* Run the game.

\*/

public void displayGame() {

java.awt.EventQueue.invokeLater(new Runnable() {

public void run() {

setVisible(true);

}

});

}

/\*\*

\* Draw the display (cards and messages).

\*/

public void repaint() {

for (int k = 0; k < board.size(); k++) {

String cardImageFileName =

imageFileName(board.cardAt(k), selections[k]);

URL imageURL = getClass().getResource(cardImageFileName);

if (imageURL != null) {

ImageIcon icon = new ImageIcon(imageURL);

displayCards[k].setIcon(icon);

displayCards[k].setVisible(true);

} else {

throw new RuntimeException(

"Card image not found: \"" + cardImageFileName + "\"");

}

}

statusMsg.setText(board.deckSize()

+ " undealt cards remain.");

statusMsg.setVisible(true);

totalsMsg.setText("You've won " + totalWins

+ " out of " + totalGames + " games.");

totalsMsg.setVisible(true);

pack();

panel.repaint();

}

/\*\*

\* Initialize the display.

\*/

private void initDisplay() {

panel = new JPanel() {

public void paintComponent(Graphics g) {

super.paintComponent(g);

}

};

// If board object's class name follows the standard format

// of ...Board or ...board, use the prefix for the JFrame title

String className = board.getClass().getSimpleName();

int classNameLen = className.length();

int boardLen = "Board".length();

String boardStr = className.substring(classNameLen - boardLen);

if (boardStr.equals("Board") || boardStr.equals("board")) {

int titleLength = classNameLen - boardLen;

setTitle(className.substring(0, titleLength));

}

// Calculate number of rows of cards (5 cards per row)

// and adjust JFrame height if necessary

int numCardRows = (board.size() + 4) / 5;

int height = DEFAULT\_HEIGHT;

if (numCardRows > 2) {

height += (numCardRows - 2) \* LAYOUT\_HEIGHT\_INC;

}

this.setSize(new Dimension(DEFAULT\_WIDTH, height));

panel.setLayout(null);

panel.setPreferredSize(

new Dimension(DEFAULT\_WIDTH - 20, height - 20));

displayCards = new JLabel[board.size()];

for (int k = 0; k < board.size(); k++) {

displayCards[k] = new JLabel();

panel.add(displayCards[k]);

displayCards[k].setBounds(cardCoords[k].x, cardCoords[k].y,

CARD\_WIDTH, CARD\_HEIGHT);

displayCards[k].addMouseListener(new MyMouseListener());

selections[k] = false;

}

replaceButton = new JButton();

replaceButton.setText("Replace");

panel.add(replaceButton);

replaceButton.setBounds(BUTTON\_LEFT, BUTTON\_TOP, 100, 30);

replaceButton.addActionListener(this);

restartButton = new JButton();

restartButton.setText("Restart");

panel.add(restartButton);

restartButton.setBounds(BUTTON\_LEFT, BUTTON\_TOP + BUTTON\_HEIGHT\_INC,

100, 30);

restartButton.addActionListener(this);

statusMsg = new JLabel(

board.deckSize() + " undealt cards remain.");

panel.add(statusMsg);

statusMsg.setBounds(LABEL\_LEFT, LABEL\_TOP, 250, 30);

winMsg = new JLabel();

winMsg.setBounds(LABEL\_LEFT, LABEL\_TOP + LABEL\_HEIGHT\_INC, 200, 30);

winMsg.setFont(new Font("SansSerif", Font.BOLD, 25));

winMsg.setForeground(Color.GREEN);

winMsg.setText("You win!");

panel.add(winMsg);

winMsg.setVisible(false);

lossMsg = new JLabel();

lossMsg.setBounds(LABEL\_LEFT, LABEL\_TOP + LABEL\_HEIGHT\_INC, 200, 30);

lossMsg.setFont(new Font("SanSerif", Font.BOLD, 25));

lossMsg.setForeground(Color.RED);

lossMsg.setText("Sorry, you lose.");

panel.add(lossMsg);

lossMsg.setVisible(false);

totalsMsg = new JLabel("You've won " + totalWins

+ " out of " + totalGames + " games.");

totalsMsg.setBounds(LABEL\_LEFT, LABEL\_TOP + 2 \* LABEL\_HEIGHT\_INC,

250, 30);

panel.add(totalsMsg);

if (!board.anotherPlayIsPossible()) {

signalLoss();

}

pack();

getContentPane().add(panel);

getRootPane().setDefaultButton(replaceButton);

panel.setVisible(true);

}

/\*\*

\* Deal with the user clicking on something other than a button or a card.

\*/

private void signalError() {

Toolkit t = panel.getToolkit();

t.beep();

}

/\*\*

\* Returns the image that corresponds to the input card.

\* Image names have the format "[Rank][Suit].GIF" or "[Rank][Suit]S.GIF",

\* for example "aceclubs.GIF" or "8heartsS.GIF". The "S" indicates that

\* the card is selected.

\*

\* @param c Card to get the image for

\* @param isSelected flag that indicates if the card is selected

\* @return String representation of the image

\*/

private String imageFileName(Card c, boolean isSelected) {

String str = "cards/";

if (c == null) {

return "cards/back1.GIF";

}

str += c.rank() + c.suit();

if (isSelected) {

str += "S";

}

str += ".GIF";

return str;

}

/\*\*

\* Respond to a button click (on either the "Replace" button

\* or the "Restart" button).

\* @param e the button click action event

\*/

public void actionPerformed(ActionEvent e) {

if (e.getSource().equals(replaceButton)) {

// Gather all the selected cards.

List<Integer> selection = new ArrayList<Integer>();

for (int k = 0; k < board.size(); k++) {

if (selections[k]) {

selection.add(new Integer(k));

}

}

// Make sure that the selected cards represent a legal replacement.

if (!board.isLegal(selection)) {

signalError();

return;

}

for (int k = 0; k < board.size(); k++) {

selections[k] = false;

}

// Do the replace.

board.replaceSelectedCards(selection);

if (board.isEmpty()) {

signalWin();

} else if (!board.anotherPlayIsPossible()) {

signalLoss();

}

repaint();

} else if (e.getSource().equals(restartButton)) {

board.newGame();

getRootPane().setDefaultButton(replaceButton);

winMsg.setVisible(false);

lossMsg.setVisible(false);

if (!board.anotherPlayIsPossible()) {

signalLoss();

lossMsg.setVisible(true);

}

for (int i = 0; i < selections.length; i++) {

selections[i] = false;

}

repaint();

} else {

signalError();

return;

}

}

/\*\*

\* Display a win.

\*/

private void signalWin() {

getRootPane().setDefaultButton(restartButton);

winMsg.setVisible(true);

totalWins++;

totalGames++;

}

/\*\*

\* Display a loss.

\*/

private void signalLoss() {

getRootPane().setDefaultButton(restartButton);

lossMsg.setVisible(true);

totalGames++;

}

/\*\*

\* Receives and handles mouse clicks. Other mouse events are ignored.

\*/

private class MyMouseListener implements MouseListener {

/\*\*

\* Handle a mouse click on a card by toggling its "selected" property.

\* Each card is represented as a label.

\* @param e the mouse event.

\*/

public void mouseClicked(MouseEvent e) {

for (int k = 0; k < board.size(); k++) {

if (e.getSource().equals(displayCards[k])

&& board.cardAt(k) != null) {

selections[k] = !selections[k];

repaint();

return;

}

}

signalError();

}

/\*\*

\* Ignore a mouse exited event.

\* @param e the mouse event.

\*/

public void mouseExited(MouseEvent e) {

}

/\*\*

\* Ignore a mouse released event.

\* @param e the mouse event.

\*/

public void mouseReleased(MouseEvent e) {

}

/\*\*

\* Ignore a mouse entered event.

\* @param e the mouse event.

\*/

public void mouseEntered(MouseEvent e) {

}

/\*\*

\* Ignore a mouse pressed event.

\* @param e the mouse event.

\*/

public void mousePressed(MouseEvent e) {

}

}

}

/\*\*

\* This is a class that tests the Card class.

\*/

public class CardTester {

/\*\*

\* The main method in this class checks the Card operations for consistency.

\* @param args is not used.

\*/

public static void main(String[] args)

{

Card queen = new Card("Queen","Spades",12);

Card ace = new Card("Ace","Diamonds",1);

Card ten = new Card("10","Clubs",10);

System.out.println("Are Aces and Queens the same?");

System.out.println(ace.matches(queen));

System.out.println();

System.out.println(ace.toString());

System.out.println(ten.toString());

System.out.println(queen.toString());

System.out.println();

System.out.println("Are Tens the same?");

System.out.println(ten.matches(ten));

}

}

import java.util.List;

import java.util.ArrayList;

import java.util.\*;

/\*\*

\* The Deck class represents a shuffled deck of cards.

\* It provides several operations including

\* initialize, shuffle, deal, and check if empty.

\*/

public class Deck

{

/\*\*

\* cards contains all the cards in the deck.

\*/

private ArrayList<Card> cards = new ArrayList<Card>();

/\*\*

\* size is the number of not-yet-dealt cards.

\* Cards are dealt from the top (highest index) down.

\* The next card to be dealt is at size - 1.

\*/

private int size;

/\*\*

\* Creates a new <code>Deck</code> instance.<BR>

\* It pairs each element of ranks with each element of suits,

\* and produces one of the corresponding card.

\* @param ranks is an array containing all of the card ranks.

\* @param suits is an array containing all of the card suits.

\* @param values is an array containing all of the card point values.

\*/

public Deck(String[] ranks, String[] suits, int[] values)

{

for(int i = 0; i < suits.length; i++)

{

for(int j = 0; j < ranks.length; j++)

{

cards.add(new Card(ranks[j],suits[i],values[j]));

size++;

}

}

shuffle();

}

/\*\*

\* Determines if this deck is empty (no undealt cards).

\* @return true if this deck is empty, false otherwise.

\*/

public boolean isEmpty()

{

if(size == 0)

{

return true;

}

else

{

return false;

}

}

/\*\*

\* Accesses the number of undealt cards in this deck.

\* @return the number of undealt cards in this deck.

\*/

public int size()

{

return size;

}

/\*\*

\* Randomly permute the given collection of cards

\* and reset the size to represent the entire deck.

\*/

public void shuffle()

{

Random rnd = new Random();

for (int i = cards.size() - 1; i >= 0; i--)

{

int index = rnd.nextInt(cards.size() - 1);

Card a = cards.get(i);

cards.set(i, cards.get(index));

cards.set(index, a);

}

}

/\*\*

\* Deals a card from this deck.

\* @return the card just dealt, or null if all the cards have been

\* previously dealt.

\*/

public Card deal()

{

if (size == 0)

{

return null;

}

else

{

size--;

return cards.get(size);

}

}

/\*\*

\* Generates and returns a string representation of this deck.

\* @return a string representation of this deck.

\*/

@Override

public String toString()

{

String rtn = "size = " + size + "\nUndealt cards: \n";

for (int k = size - 1; k >= 0; k--) {

rtn = rtn + cards.get(k);

if (k != 0) {

rtn = rtn + ", ";

}

if ((size - k) % 2 == 0) {

// Insert carriage returns so entire deck is visible on console.

rtn = rtn + "\n";

}

}

rtn = rtn + "\nDealt cards: \n";

for (int k = cards.size() - 1; k >= size; k--) {

rtn = rtn + cards.get(k);

if (k != size) {

rtn = rtn + ", ";

}

if ((k - cards.size()) % 2 == 0) {

// Insert carriage returns so entire deck is visible on console.

rtn = rtn + "\n";

}

}

rtn = rtn + "\n";

return rtn;

}

}

/\*\*

\* This is a class that tests the Deck class.

\*/

public class DeckTester {

/\*\*

\* The main method in this class checks the Deck operations for consistency.

\* @param args is not used.

\*/

public static void main(String[] args)

{

String[] suits = {"Hearts","Diamonds","Spades","Clubs"};

int[] values= {1,2,3,4,5,6,7,8,9,10,11,12,13};

String[] ranks = {"Ace","2","3","4","5","6","7","8","9","10","Jack","Queen","King"};

Deck cards = new Deck(ranks,suits,values);

System.out.println("Is the deck full?");

System.out.println(cards.isEmpty());

System.out.println();

System.out.println("How large is the deck?");

System.out.println(cards.size());

cards.shuffle();

cards.deal();

System.out.println();

System.out.println(cards.toString());

}

}

import java.util.List;

import java.util.ArrayList;

import java.util.\*;

/\*\*

\* The ElevensBoard class represents the board in a game of Elevens.

\*/

public class ElevensBoard extends Board

{

/\*\*

\* The size (number of cards) on the board.

\*/

private static final int BOARD\_SIZE = 9;

/\*\*

\* The ranks of the cards for this game to be sent to the deck.

\*/

private static final String[] RANKS =

{"ace", "2", "3", "4", "5", "6", "7", "8", "9", "10", "jack", "queen", "king"};

/\*\*

\* The suits of the cards for this game to be sent to the deck.

\*/

private static final String[] SUITS =

{"spades", "hearts", "diamonds", "clubs"};

/\*\*

\* The values of the cards for this game to be sent to the deck.

\*/

private static final int[] POINT\_VALUES =

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 0, 0};

/\*\*

\* Flag used to control debugging print statements.

\*/

private static final boolean I\_AM\_DEBUGGING = false;

/\*\*

\* Creates a new <code>ElevensBoard</code> instance.

\*/

public ElevensBoard()

{

super(BOARD\_SIZE, RANKS, SUITS, POINT\_VALUES);

}

/\*\*

\* Determines if the selected cards form a valid group for removal.

\* In Elevens, the legal groups are (1) a pair of non-face cards

\* whose values add to 11, and (2) a group of three cards consisting of

\* a jack, a queen, and a king in some order.

\* @param selectedCards the list of the indexes of the selected cards.

\* @return true if the selected cards form a valid group for removal;

\* false otherwise.

\*/

@Override

public boolean isLegal(List<Integer> selectedCards)

{

if (containsPairSum11(selectedCards) == true || containsJQK(selectedCards) == true)

{

return true;

}

else

{

return false;

}

}

/\*\*

\* Determine if there are any legal plays left on the board.

\* In Elevens, there is a legal play if the board contains

\* (1) a pair of non-face cards whose values add to 11, or (2) a group

\* of three cards consisting of a jack, a queen, and a king in some order.

\* @return true if there is a legal play left on the board;

\* false otherwise.

\*/

@Override

public boolean anotherPlayIsPossible()

{

if (((containsPairSum11(cardIndexes())) == true) || ((containsJQK(cardIndexes())) == true))

{

return true;

}

else

{

return false;

}

}

/\*\*

\* Check for an 11-pair in the selected cards.

\* @param selectedCards selects a subset of this board. It is list

\* of indexes into this board that are searched

\* to find an 11-pair.

\* @return true if the board entries in selectedCards

\* contain an 11-pair; false otherwise.

\*/

private boolean containsPairSum11(List<Integer> selectedCards)

{

int i = 0;

int j = 1;

while (i < selectedCards.size())

{

while (j < selectedCards.size())

{

int sum = cardAt(selectedCards.get(i)).pointValue() + cardAt(selectedCards.get(j)).pointValue();

if (sum == 11)

{

int a = selectedCards.get(i);

int b = selectedCards.get(j);

selectedCards.clear();

selectedCards.add(a);

selectedCards.add(b);

return true;

}

j++;

}

i++;

j = i + 1;

}

return false;

}

/\*\*

\* Check for a JQK in the selected cards.

\* @param selectedCards selects a subset of this board. It is list

\* of indexes into this board that are searched

\* to find a JQK group.

\* @return true if the board entries in selectedCards

\* include a jack, a queen, and a king; false otherwise.

\*/

private boolean containsJQK(List<Integer> selectedCards)

{

ArrayList<String> ranks = new ArrayList<String>();

int i = 0;

for (i = 0; i < selectedCards.size(); i++)

{

ranks.add(cardAt(selectedCards.get(i)).rank());

}

if ((ranks.contains("jack") && (ranks.contains("king") && (ranks.contains("queen")))))

{

return true;

}

else

{

return false;

}

}

}

/\*\*

\* This is a class that plays the GUI version of the Elevens game.

\* See accompanying documents for a description of how Elevens is played.

\*/

public class ElevensGUIRunner {

/\*\*

\* Plays the GUI version of Elevens.

\* @param args is not used.

\*/

public static void main(String[] args) {

Board board = new ElevensBoard();

CardGameGUI gui = new CardGameGUI(board);

gui.displayGame();

}

}

import java.util.\*;

/\*\*

\* This class provides a convenient way to test shuffling methods.

\* @author Anika Jallipalli

\* @version 4/11/2020

\*/

public class Shuffler

{

/\*\*

\* The number of consecutive shuffle steps to be performed in each call

\* to each sorting procedure.

\*/

private static final int SHUFFLE\_COUNT = 16;

/\*\*

\* The number of values to shuffle.

\*/

private static final int VALUE\_COUNT = 4;

/\*\*

\* Tests shuffling methods.

\* @param args is not used.

\*/

public static void main(String[] args)

{

System.out.println("Results of " + SHUFFLE\_COUNT +

" consecutive perfect shuffles:");

int[] values1 = new int[VALUE\_COUNT];

for (int i = 0; i < values1.length; i++) {

values1[i] = i;

}

for (int j = 1; j <= SHUFFLE\_COUNT; j++) {

perfectShuffle(values1);

System.out.print(" " + j + ":");

for (int k = 0; k < values1.length; k++) {

System.out.print(" " + values1[k]);

}

System.out.println();

}

System.out.println();

System.out.println("Results of " + SHUFFLE\_COUNT +

" consecutive efficient selection shuffles:");

int[] values2 = new int[VALUE\_COUNT];

for (int i = 0; i < values2.length; i++) {

values2[i] = i;

}

for (int j = 1; j <= SHUFFLE\_COUNT; j++) {

selectionShuffle(values2);

System.out.print(" " + j + ":");

for (int k = 0; k < values2.length; k++) {

System.out.print(" " + values2[k]);

}

System.out.println();

}

System.out.println();

System.out.println(flip());

System.out.println();

int[] a = {1,2,3,4};

int[] b = {4,5,2,1};

arePermutations(a,b);

}

/\*\*

\* Apply a "perfect shuffle" to the argument.

\* The perfect shuffle algorithm splits the deck in half, then interleaves

\* the cards in one half with the cards in the other.

\* @param values is an array of integers simulating cards to be shuffled.

\*/

public static void perfectShuffle(int[] values)

{

int k = 0;

int[] shuffled = new int[values.length];

for(int i=0; i < (values.length)/2; i++)

{

shuffled[k] = values[i];

k +=2;

}

k = 1;

for(int i= (values.length)/2; i < values.length; i++)

{

shuffled[k] = values[i];

k +=2;

}

for(int i = 0; i < shuffled.length; i++)

{

values[i] = shuffled[i];

}

}

/\*\*

\* Apply an "efficient selection shuffle" to the argument.

\* The selection shuffle algorithm conceptually maintains two sequences

\* of cards: the selected cards (initially empty) and the not-yet-selected

\* cards (initially the entire deck). It repeatedly does the following until

\* all cards have been selected: randomly remove a card from those not yet

\* selected and add it to the selected cards.

\* An efficient version of this algorithm makes use of arrays to avoid

\* searching for an as-yet-unselected card.

\* @param values is an array of integers simulating cards to be shuffled.

\*/

public static void selectionShuffle(int[] values)

{

Random rnd = new Random();

int[] shuffled = new int[values.length];

for (int i = values.length - 1; i > 0; i--)

{

int index = rnd.nextInt(i + 1);

// Simple swap

int a = values[index];

values[index] = values[i];

values[i] = a;

}

}

public static String flip()

{

Random rnd = new Random();

String side;

int n = rnd.nextInt(3);

if(n==0 | n==1)

{

side = "heads";

}

else

{

side = "tails";

}

return side;

}

public static void arePermutations(int[]a, int[] b)

{

boolean[] perm = new boolean[a.length];

if(a.length == b.length)

{

for(int i = 0; i < a.length; i++)

{

perm[i] = false;

for(int j = 0; j < b.length; j++)

{

if(a[i] == b[j])

{

perm[i] = true;

}

}

}

boolean isPerm = true;

for(int i=0; i < perm.length;i++)

{

if(perm[i] != true)

{

isPerm = false;

break;

}

}

if(isPerm == true)

{

System.out.println("They are permutations.");

}

else

{

System.out.println("There are not permutations.");

}

}

}

}

import java.util.List;

import java.util.ArrayList;

/\*\*

\* This class represents a Board that can be used in a collection

\* of solitaire games similar to Elevens. The variants differ in

\* card removal and the board size.

\* @author Anika Jallipalli

\* @version 4/11/2020

\*/

public abstract class Board

{

/\*\*

\* The cards on this board.

\*/

private Card[] cards;

/\*\*

\* The deck of cards being used to play the current game.

\*/

private Deck deck;

/\*\*

\* Flag used to control debugging print statements.

\*/

private static final boolean I\_AM\_DEBUGGING = false;

/\*\*

\* Creates a new <code>Board</code> instance.

\* @param size the number of cards in the board

\* @param ranks the names of the card ranks needed to create the deck

\* @param suits the names of the card suits needed to create the deck

\* @param pointValues the integer values of the cards needed to create

\* the deck

\*/

public Board(int size, String[] ranks, String[] suits, int[] pointValues)

{

cards = new Card[size];

deck = new Deck(ranks, suits, pointValues);

if (I\_AM\_DEBUGGING)

{

System.out.println(deck);

System.out.println("----------");

}

dealMyCards();

}

/\*\*

\* Start a new game by shuffling the deck and

\* dealing some cards to this board.

\*/

public void newGame()

{

deck.shuffle();

dealMyCards();

}

/\*\*

\* Accesses the size of the board.

\* Note that this is not the number of cards it contains,

\* which will be smaller near the end of a winning game.

\* @return the size of the board

\*/

public int size()

{

return cards.length;

}

/\*\*

\* Determines if the board is empty (has no cards).

\* @return true if this board is empty; false otherwise.

\*/

public boolean isEmpty()

{

for (int k = 0; k < cards.length; k++)

{

if (cards[k] != null)

{

return false;

}

}

return true;

}

/\*\*

\* Deal a card to the kth position in this board.

\* If the deck is empty, the kth card is set to null.

\* @param k the index of the card to be dealt.

\*/

public void deal(int k)

{

cards[k] = deck.deal();

}

/\*\*

\* Accesses the deck's size.

\* @return the number of undealt cards left in the deck.

\*/

public int deckSize()

{

return deck.size();

}

/\*\*

\* Accesses a card on the board.

\* @return the card at position k on the board.

\* @param k is the board position of the card to return.

\*/

public Card cardAt(int k)

{

return cards[k];

}

/\*\*

\* Replaces selected cards on the board by dealing new cards.

\* @param selectedCards is a list of the indices of the

\* cards to be replaced.

\*/

public void replaceSelectedCards(List<Integer> selectedCards)

{

for (Integer k : selectedCards)

{

deal(k.intValue());

}

}

/\*\*

\* Gets the indexes of the actual (non-null) cards on the board.

\*

\* @return a List that contains the locations (indexes)

\* of the non-null entries on the board.

\*/

public List<Integer> cardIndexes()

{

List<Integer> selected = new ArrayList<Integer>();

for (int k = 0; k < cards.length; k++)

{

if (cards[k] != null)

{

selected.add(new Integer(k));

}

}

return selected;

}

/\*\*

\* Generates and returns a string representation of this board.

\* @return the string version of this board.

\*/

public String toString()

{

String s = "";

for (int k = 0; k < cards.length; k++)

{

s = s + k + ": " + cards[k] + "\n";

}

return s;

}

/\*\*

\* Determine whether or not the game has been won,

\* i.e. neither the board nor the deck has any more cards.

\* @return true when the current game has been won;

\* false otherwise.

\*/

public boolean gameIsWon()

{

if (deck.isEmpty())

{

for (Card c : cards)

{

if (c != null)

{

return false;

}

}

return true;

}

return false;

}

/\*\*

\* Method to be completed by the concrete class that determines

\* if the selected cards form a valid group for removal.

\* @param selectedCards the list of the indices of the selected cards.

\* @return true if the selected cards form a valid group for removal;

\* false otherwise.

\*/

public abstract boolean isLegal(List<Integer> selectedCards);

/\*\*

\* Method to be completed by the concrete class that determines

\* if there are any legal plays left on the board.

\* @return true if there is a legal play left on the board;

\* false otherwise.

\*/

public abstract boolean anotherPlayIsPossible();

/\*\*

\* Deal cards to this board to start the game.

\*/

private void dealMyCards()

{

for (int k = 0; k < cards.length; k++)

{

cards[k] = deck.deal();

}

}

}