AP® Computer Science A

Elevens Lab

Student Guide

CollegeBoard

Elevens Lab Student Guide

Introduction

The following activities are related to a simple solitaire game called Elevens. You will learn the rules of Elevens, and will be able to play it by using the supplied Graphical User Interface (GUI) shown at the right. You will learn about the design and the Object Oriented Principles that suggested that design. You will also implement much of the code.
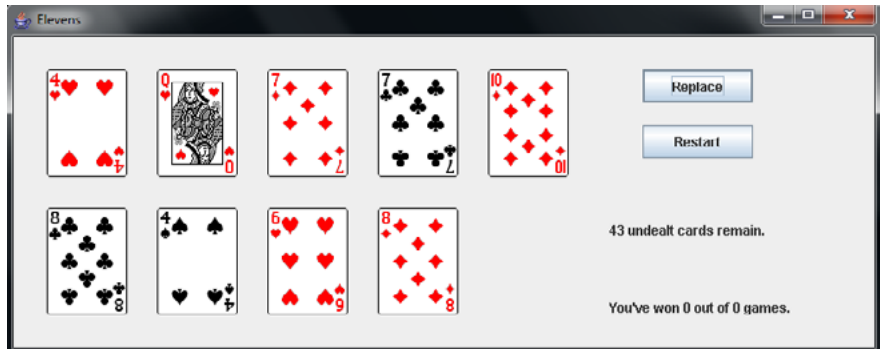
Table of Contents

Activity 3: Shuffling the Cards in a Deck

Introduction:

Think about how you shuffle a deck of cards by hand. How well do you think it randomizes the cards in the deck?

Exploration:

We now consider the *shuffling* of a deck, that is, the *permutation* of its cards into a random-looking sequence. A requirement of the shuffling procedure is that any particular permutation has just as much chance of occurring as any other. We will be using the `Math.random` method to generate random numbers to produce these permutations.

Several ideas for designing a shuffling method come to mind. We will consider two:

Perfect Shuffle

Card players often shuffle by splitting the deck in half and then interleaving the two half-decks, as shown below.
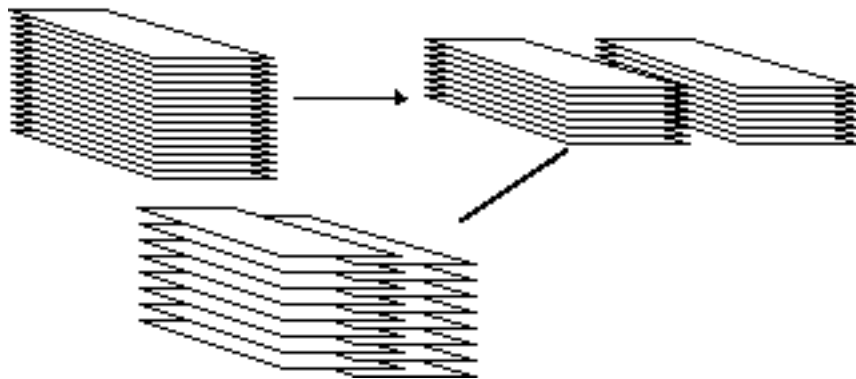
This procedure is called a *perfect shuffle* if the interleaving alternates between the two half-decks. Unfortunately, the perfect shuffle comes nowhere near generating all possible deck permutations. In fact, eight shuffles of a 52-card deck return the deck to its original state!

Consider the following "perfect shuffle" algorithm that starts with an array named `cards` that contains 52 cards and creates an array named `shuffled`.

> Initialize `shuffled` to contain 52 "empty" elements.
> Set `k` to 0.
> For `j` = 0 to 25,
>   − Copy `cards[j]` to `shuffled[k]`;
>   − Set `k` to `k+2`.
> Set `k` to 1.
> For `j` = 26 to 51,
>   − Copy `cards[j]` to `shuffled[k]`;
>   − Set `k` to `k+2`.

This approach moves the first half of `cards` to the even index positions of `shuffled`, and it moves the second half of `cards` to the odd index positions of `shuffled`.

The above algorithm shuffles 52 cards. If an odd number of cards is shuffled, the array `shuffled` has one more even-indexed position than odd-indexed positions. Therefore, the first loop must copy one more card than the second loop does. This requires rounding up when calculating the index of the middle of the deck. In other words, in the first loop `j` must go up to `(cards.length + 1) / 2`, exclusive, and in the second loop `j` most begin at `(cards.length + 1) / 2`.

Selection Shuffle

Consider the following algorithm that starts with an array named `cards` that contains 52 cards and creates an array named `shuffled`. We will call this algorithm the "selection shuffle."

> Initialize `shuffled` to contain 52 "empty" elements.
> Then for `k` = 0 to 51,
>   − Repeatedly generate a random integer `j` between 0 and 51, inclusive until `cards[j]` contains a card (not marked as empty);
>   − Copy `cards[j]` to `shuffled[k]`;
>   − Set `cards[j]` to empty.

This approach finds a suitable card for the kth position of the deck. Unsuitable candidates are any cards that have already been placed in the deck.

While this is a more promising approach than the perfect shuffle, its big defect is that it runs too slowly. Every time an empty element is selected, it has to loop again. To determine the last element of `shuffled` requires an average of 52 calls to the random number generator.

A better version, the "efficient selection shuffle," works as follows:

> For k = 51 down to 1,
> - Generate a random integer r between 0 and k, inclusive;
> - Exchange cards[k] and cards[r].

This has the same structure as selection sort:

> For k = 51 down to 1,
> - Find r, the position of the largest value among cards[0] through cards[k];
> - Exchange cards[k] and cards[r].

The selection shuffle algorithm does not require to a loop to find the largest (or smallest) value to swap, so it works quickly.

Exercises:

1. Use the file Shuffler.java, found in the **Activity3 Starter Code**, to implement the perfect shuffle and the efficient selection shuffle methods as described in the **Exploration** section of this activity. You will be shuffling arrays of integers.

2. Shuffler.java also provides a main method that calls the shuffling methods. Execute the main method and inspect the output to see how well each shuffle method actually randomizes the array elements. You should execute main with different values of SHUFFLE_COUNT and VALUE_COUNT.

Questions:

1. Write a static method named flip that simulates a flip of a weighted coin by returning either "heads" or "tails" each time it is called. The coin is twice as likely to turn up heads as tails. Thus, flip should return "heads" about twice as often as it returns "tails."

2. Write a static method named arePermutations that, given two int arrays of the same length but with no duplicate elements, returns true if one array is a permutation of the other (i.e., the arrays differ only in how their contents are arranged). Otherwise, it should return false.

3. Suppose that the initial contents of the values array in Shuffler.java are {1, 2, 3, 4}. For what sequence of random integers would the efficient selection shuffle change values to contain {4, 3, 2, 1}?

Activity 4: Adding a `Shuffle` Method to the `Deck` Class

Introduction:

You implemented a `Deck` class in Activity 2. This class should be complete except for the `shuffle` method. You also implemented a `DeckTester` class that you used to test your incomplete `Deck` class.

In Activity 3, you implemented methods in the `Shuffler` class, which shuffled integers.

Now you will use what you learned about shuffling in Activity 3 to implement the `Deck shuffle` method.

Exercises:

1. The file `Deck.java,` found in the **Activity4 Starter Code** folder, is a correct solution from Activity 2. Complete the `Deck` class by implementing the `shuffle` method. Use the efficient selection shuffle algorithm from Activity 3.

   Note that the `Deck` constructor creates the deck and then calls the `shuffle` method. The `shuffle` method also needs to reset the value of `size` to indicate that all of the cards can be dealt again.

2. The `DeckTester.java` file, found in the **Activity4 Starter Code** folder, provides a basic set of `Deck` tests. It is similar to the `DeckTester` class you might have written in Activity 2.