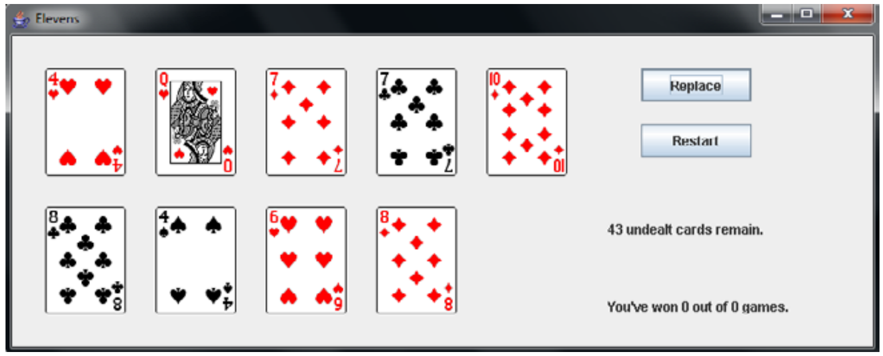
AP® Computer Science A
Elevens Lab
Student Guide

The AP Program wishes to acknowledge and thank the following individuals for their contributions in developing this lab and the accompanying documentation.

Michael Clancy: University of California at Berkeley
Robert Glen Martin: School for the Talented and Gifted in Dallas, TX
Judith Hromcik: School for the Talented and Gifted in Dallas, TX

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 1: Design and Create a `Card` Class

Introduction:

In this activity, you will complete a `Card` class that will be used to create card objects.

Think about card games you've played. What kinds of information do these games require a card object to "know"? What kinds of operations do these games require a card object to provide?

Exploration:

Now think about implementing a class to represent a playing card. What instance variables should it have? What methods should it provide? Discuss your ideas for this `Card` class with classmates.

Read the partial implementation of the `Card` class available in the **Activity1 Starter Code** folder. As you read through this class, you will notice the use of the `@Override` annotation before the `toString` method. The Java `@Override` annotation can be used to indicate that a method is intended to override a method in a superclass. In this example, the `Object` class's `toString` method is being overridden in the `Card` class. If the indicated method doesn't override a method, then the Java compiler will give an error message.

Here's a situation where this facility comes in handy. Programmers new to Java often encounter problems matching headings of overridden methods to the superclass's original method heading. For example, in the `Weight` class below, the `tostring` method is intended to be invoked when `toString` is called for a `Weight` object.

```
public class Weight {
   private int pounds;
   private int ounces;
       ...
   public String tostring(String str) {
      return this.pounds + " lb.  " + this.ounces + " oz.";
   }
       ...
}
```

Unfortunately, this doesn't work; the `tostring` method given above has a different name and a different signature from the `Object` class's `toString` method. The correct version below has the correct name `toString` and no parameter:

```
   public String toString() {
      return this.pounds + " lb.  " + this.ounces + " oz.";
   }
```

The `@Override` annotation would cause an error message for the first `tostring` version to alert the programmer of the errors.

Exercises:

1. Complete the implementation of the provided `Card` class. You will be required to complete:

    a. a constructor that takes two `String` parameters that represent the card's rank and suit, and an `int` parameter that represents the point value of the card;

    b. accessor methods for the card's rank, suit, and point value;

    c. a method to test equality between two card objects; and

    d. the `toString` method to create a `String` that contains the rank, suit, and point value of the card object. The string should be in the following format:

        rank `of` *suit* `(point value =` *pointValue*`)`

2. Once you have completed the `Card` class, find the `CardTester.java` file in the **Activity1 Starter Code** folder. Create three `Card` objects and test each method for each `Card` object.

Activity 2: Initial Design of a `Deck` Class

Introduction:

Think about a deck of cards. How would you describe a deck of cards? When you play card games, what kinds of operations do these games require a deck to provide?

Exploration:

Now consider implementing a class to represent a deck of cards. Describe its instance variables and methods, and discuss your design with a classmate.

Read the partial implementation of the `Deck` class available in the **Activity2 Starter Code** folder. This file contains the instance variables, constructor header, and method headers for a `Deck` class general enough to be useful for a variety of card games. Discuss the `Deck` class with your classmates; in particular, make sure you understand the role of each of the parameters to the `Deck` constructor, and of each of the private instance variables in the `Deck` class.

Exercises:

1. Complete the implementation of the `Deck` class by coding each of the following:

    - `Deck` constructor — This constructor receives three arrays as parameters. The arrays contain the ranks, suits, and point values for each card in the deck. The constructor creates an `ArrayList`, and then creates the specified cards and adds them to the list. For example, if `ranks = {"A", "B", "C"}`, `suits = {"Giraffes", "Lions"}`, and `values = {2,1,6}`, the constructor would create the following cards:

      ```
      ["A", "Giraffes", 2], ["B", "Giraffes", 1], ["C", "Giraffes", 6],
      ["A", "Lions", 2], ["B", "Lions", 1], ["C", "Lions", 6]
      ```

      and would add each of them to `cards`. The parameter `size` would then be set to the size of `cards`, which in this example is 6.

      Finally, the constructor should shuffle the deck by calling the `shuffle` method. Note that you will not be implementing the `shuffle` method until Activity 4.

    - `isEmpty` — This method should return `true` when the size of the deck is `0`; `false` otherwise.

    - `size` — This method returns the number of cards in the deck that are left to be dealt.

- `deal` — This method "deals" a card by removing a card from the deck and returning it, if there are any cards in the deck left to be dealt. It returns `null` if the deck is empty. There are several ways of accomplishing this task. Here are two possible algorithms:
  Algorithm 1: Because the cards are being held in an `ArrayList`, it would be easy to simply call the `List` method that removes an object at a specified index, and return that object. Removing the object from the end of the list would be more efficient than removing it from the beginning of the list. Note that the use of this algorithm also requires a separate "discard" list to keep track of the dealt cards. This is necessary so that the dealt cards can be reshuffled and dealt again.
  Algorithm 2: It would be more efficient to leave the cards in the list. Instead of removing the card, simply decrement the size instance variable and then return the card at `size`. In this algorithm, the `size` instance variable does double duty; it determines which card to "deal" and it also represents how many cards in the deck are left to be dealt. **This is the algorithm that you should implement.**

2. Once you have completed the `Deck` class, find `DeckTester.java` file in the **Activity2 Starter Code** folder. Add code in the `main` method to create three `Deck` objects and test each method for each `Deck` object.

Questions:

1. Explain in your own words the relationship between a `deck` and a `card`.

2. Consider the deck initialized with the statements below. How many cards does the deck contain?

```
String[] ranks = {"jack", "queen", "king"};
String[] suits = {"blue", "red"};
int[] pointValues = {11, 12, 13};
Deck d = new Deck(ranks, suits, pointValues);
```

3. The game of Twenty-One is played with a deck of 52 cards. Ranks run from ace (highest) down to 2 (lowest). Suits are spades, hearts, diamonds, and clubs as in many other games. A face card has point value 10; an ace has point value 11; point values for 2, …, 10 are 2, …, 10, respectively. Specify the contents of the `ranks`, `suits`, and `pointValues` arrays so that the statement

```
Deck d = new Deck(ranks, suits, pointValues);
```

   initializes a deck for a Twenty-One game.

4. Does the order of elements of the `ranks`, `suits`, and `pointValues` arrays matter?

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?

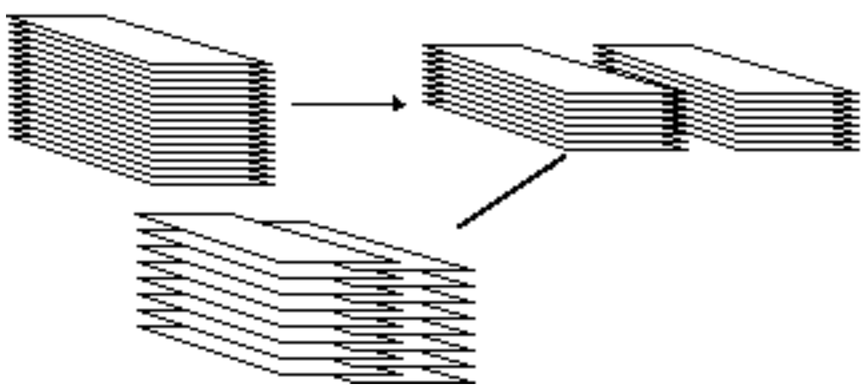
Exploration:

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 k[th] 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.