# CCPS506 Assignment – Fall 2019

For this assignment, you will solve the same problem for **two** of the four languages we studied this semester. One of the languages chosen must be functional (Elixir or Haskell). You are welcome and encouraged to submit assignments in more than two languages! If you do, the best two will be used to for your assignment mark.

In addition to the general requirements of the assignment, each language comes with its own slightly modified set of language-specific constraints. These specify the format of the input and output, as well as submission instructions for each language. Aside from these, anything you do inside your program is up to you. Use as many helper functions as you want, use any and all syntax you find useful whether we covered it in class or not, with one caveat: you may not use any functionality that is not part of the base installation of each language. No 3rd party libraries.

All assignment submissions are due at the same time: **Tuesday, Dec 3, 11:59pm**

## General assignment description:

For this assignment, you will write a program that deals and evaluates two five-card poker hands and chooses a winner. A description of different hands and their ranking can be seen here:

**https://www.cardplayer.com/rules-of-poker/hand-rankings**

The input to your program is a permutation of the integers 1-52 that represents a ***shuffling*** of a standard deck of cards. The order of suits in an unshuffled deck are Clubs, Diamonds, Hearts, Spades. Within each suit, the ranks are ordered from 2-10, Jack, Queen, King, Ace. Thus, an input array that started with the integers [38, 48, 11, 6, …] would represent King of Hearts, 10 of Spades, Queen of Clubs, 7 of Clubs, and so on.

Your program will accept this permutation array as input and use it to deal two poker hands of 5 cards each in an alternating fashion. I.e., the first card goes to hand 1, the second card goes to hand 2, the third card goes to hand 1, fourth to hand 2, etc. until each hand has 5 cards.

Once dealt, your program will analyze each hand according to the rules from the website above and decide a winner. Your program will return the actual winning hand in a specific form that differs for each language. There will be no ties. To resolve what might otherwise be a tie in traditional poker, use the suit ordering of Clubs < Diamonds < Hearts < Spades. That is, if both hands only have a high card of 10, the hand whose 10 is from a higher suit would be considered the winner.

For all languages, your program will be driven at the top level by a function/method called **`deal`** that accepts the permutation array described above as input. This function will return the winning hand based on the format described for each language below. Anything else you do inside this **`deal`** function is completely up to you.

**Specific Language Requirements:**

### 1) Smalltalk requirements:

You must create a class called **Poker**, with a *class* method called **deal:** that accepts a simple integer array as input and returns the winning hand. Its usage will be as follows:

```
winner := Poker deal: #(3 5 16 34 12 18 …).
```

The winning hand will be returned as an array of strings where each element is a concatenation of the rank and the suit. The suit must be capitalized, and the order doesn't matter. Face cards are represented numerically. 11=Jack, 12=Queen, 13=King. An Ace is represented numerically as 1. For example:

```
#('2H' '2S' '1S' '2C' '13C').
```

### 2) Elixir requirements:

In a single Elixir file called Poker.ex, define a module called **Poker**, with a function called **deal** that accepts a list of integers as an argument and returns the winning hand. Its usage will be as follows:

```
winner = Poker.deal [3, 5, 16, 34, 12, 18, …]
```

The winning hand will be returned as a list of strings where each element is a concatenation of the rank and the suit. The suit must be capitalized, and the order doesn't matter. Face cards are represented numerically. 11=Jack, 12=Queen, 13=King. An Ace is represented numerically as 1. For example:

```
["2H", "2S", "1S", "2C", "13C"]
```

### 3) Haskell requirements:

In a single Haskell file called Poker.hs, define a module called **Poker**, with a function called **deal** that accepts a list of integers as input and returns the winning hand. Its usage will be as follows:

```
winner = Poker.deal [3, 5, 16, 34, 12, 18, …]
```

The winning hand will be returned as a list of strings where each element is a concatenation of the rank and the suit. The suit must be capitalized, and the order doesn't matter. Face cards are represented numerically. 11=Jack, 12=Queen, 13=King. An Ace is represented numerically as 1. For example:

```
["2H", "2S", "1S", "2C", "13C"]
```

### 4) Rust requirements:

In a single Rust file called Poker.rs, write a function called **deal** that accepts an array of integers as input and returns the winning hand. Its usage will be as follows:

```
let perm = [3, 5, 16, 34, 12, 18, …]
winner = deal(perm)
```

The winning hand will be returned as an array of strings where each element is a concatenation of the rank and the suit. The suit must be capitalized, and the order doesn't matter. Face cards are represented numerically. 11=Jack, 12=Queen, 13=King. An Ace is represented numerically as 1. For example:

```
["2H", "2S", "1S", "2C", "13C"]
```

## Testing

I will be testing your assignments using a large and comprehensive set of input permutations representing any and all possible hands that could be dealt. It is up to you to test your program and ensure it produces the correct winner under all conditions.

For each language, I will provide a handful of simple test cases (probably after reading week) for you to try out and ensure your program is working properly. This will also ensure that you've got the input and output format defined correctly. You may assume that the input is always valid. That is, I will always pass a correctly formed permutation of the integers 1-52. You are not expected to validate or error check the input.

## Submission

Submission for Elixir, Haskell, and Rust is very straightforward. Simply submit your source file (Poker.ex, Poker.hs, or Poker.rs) on D2L. For Smalltalk, I ask that you archive (zip/rar/7z whatever) your entire Pharo image and submit that on D2L. This way I can simply launch your image and the Poker class will be present and defined correctly.