In this project, we will be starting the process of writing a program that allows the user to play the card game Blackjack. In the first part of the project, EACH PERSON will write the foundational classes of `Card`, `Deck` and `Shoe` (described in detail below). In the second part of the project, you will be working in teams to design, code and test the remaining classes (`Hand`, `Dealer`, `Player`, and `Game`).

If you aren't familiar with cards in general or do not know how to play the game of Blackjack, don't worry! We will provide help along the way as well as a complete set of the rules of the game as we will be playing it (as with most card games, there are *many* variations on Blackjack, so we will be clarifying the rules we will be using).

IMPORTANT: This set of programs will be different than those we have done this year. *YOU are responsible for deciding what to test and how*. You will not be given testers to use. Instead, you will need to write your own main methods that use and test the classes you create. When you turn in your files, they will be run through a group of testers that the instructors have written. If all of the tests do not pass, your score on the assignment WILL BE affected. **Projects will be graded on the first submission.** If you choose to correct/update your code based on comments from your instructor, your score on the project will not change. SO, TEST YOUR CODE COMPLETELY AND THOROUGHLY BEFORE YOU TURN IT IN. You want it to pass first time out.

Here is what you need to create and turn in. All files are to be submitted in a single ZIP file. All given class names, method names and variable names MUST be used and CASE MATTERS (look for items in `courier font` to see names you must use). Failure to use the required names means your program WILL FAIL the tester.

Card.java and CardTester.java

A `Card` is the fundamental unit of play for all card games and is used directly and indirectly by every other class in such applications. As such, it is arguably the most important piece of code in this project, and it should be very flexible (so that we don't have to rewrite the `Card` class every time we want to create a different card game).

- A `Card` has three fundamental traits:
  - a `rank` (String which indicates the card value. Possible values are: `Ace, Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Jack, Queen, King`)
  - a `suit` (String indicating one of four suits in a standard deck of cards. The suits are: `Hearts, Diamonds, Spades, Clubs`)
  - a `value` (integer, the value assigned to a card can vary between card games. For our purposes, Aces will have an initial `value` of 11, face cards (Jacks, Queens and Kings) will have a `value` of 10 and all other cards have a `value` that matches its `rank` (i.e. a Two of Hearts has a value of 2).

- The `Card` class has a constructor. The constructor should take TWO inputs: the `Card rank` and the `Card suit`. The `value` of the `Card` object should be determined and assigned by the constructor.

- Since Aces have special properties in the game of BlackJack, we need a couple of special methods to deal with them.
    - `isAce` – takes no inputs, returns a `boolean` value that indicates whether a particular `Card` is an Ace (`true` means the `Card` is an Ace)

    - `setAceValue(boolean setToOne)` – Although Aces start with a `value` of 11, they can switch to a `value` of 1 in some cases. This method should start by ensuring the `Card` is actually an Ace. If it is not an Ace, the method simply exits. For Aces, the method checks the input, `setToOne`. If `setToOne` is `true`, the `value` of the `Card` should be set to 1. The `Card` `value` should be set to 11 otherwise.

- Finally, we should have getters for the `suit` (`getSuit()`), `rank` (`getRank()`) and `value` (`getValue()`) of a `Card` and a `toString()` the prints the `Card` details like this: "Ace of Hearts"

You should write the `Card` class alongside a thorough runner that tests every method of the class to ensure it works well.

Deck.java and DeckTester.java

A `Deck` contains a group of `Cards`. The standard `Deck` consists of 52 `Cards`. There are thirteen `Cards` (Ace, 2, 3, …,9, 10, J, Q, K) in each of the 4 suits.

- We will use an `ArrayList` to implement our `Deck`.

- The `Deck` class has a constructor. Creating a `Deck` should result in the creation of one and only one of each possible `Card` (52 total). You will need to consider how to code this constructor to ensure all necessary `Cards` get created (and not duplicated). As `Cards` are created, they should be added to our `ArrayList`. **DO NOT SHUFFLE the `Deck` in the constructor.** Hint: a thoughtful use of some helper arrays here can avoid a nightmare of if statements or worse…

- Other objects will be making use of our `Deck`, so a method `getDeck()` will be needed. This method returns a reference to the `ArrayList` of `Cards` contained in the `Deck`.

- Finally, we need to be able to shuffle our `Deck`. Write a method named `shuffleDeck()` that accomplishes this task. The method should loop through the `ArrayList` and for each `Card`, swap the `Card` in the current position with another `Card` found at another randomly determined location in the list.

- Although not required for the program, a `toString()` method that prints the value of each `Card` in your `Deck` can be helpful for testing purposes

You should write the `Deck` class alongside a thorough runner that tests every method of the class to ensure it works well.

Shoe.java and ShoeTester.java

Originally, Blackjack was played with a single deck of cards. But savvy card players soon learned they could win more often by "counting cards." These players would keep track of which cards had been played already, and, as a result, they were able to make more accurate guesses about what cards they might be dealt in later hands. This is a huge advantage to the player. To counteract this, most blackjack games today use a "shoe". A shoe is a device that holds multiple decks of cards.

- Like a `Deck`, a `Shoe` should be implemented as an `ArrayList` of `Card` objects.
- The `Shoe` constructor should take a single integer input that indicates how many `Decks` of `Cards` should be placed in the `Shoe`. The constructor should create the appropriate number of `Decks` and add the `Cards` from each to the `Shoe's ArrayList` of `Cards`. **DO NOT SHUFFLE the `Shoe` in the constructor.**
- The dealer uses the `Shoe` to distribute `Cards` to the players, so the `Shoe` will need a `dealCard()` method that removes the first `Card` from the `Shoe's Arraylist` and returns it.
- Like the `Deck` method, we need a way to mix the `Cards` in the `Shoe`. Using your `shuffleDeck()` method from the `Deck` class as a model, create a similar method named `shuffleShoe()` that shuffles all of the `Cards` in the `Shoe`.
- Even with a `Shoe` that holds multiple `Decks`, we will run out of cards eventually and need to start over with a new `Shoe`. To help us track that situation, we'll create a method `getNumCardsInShoe()`. This method takes no inputs and simply returns an integer indicating how many `Cards` remain in the `Shoe`.

You should write the `Shoe` class alongside a thorough runner that tests every method of the class to ensure it works well.