# CSC 150: Lab 4 – Hit the Deck!
# Thursday, April 21, 2016

## Objectives

- More practice with **information hiding**. You can have vastly different *implementations* (how the method works) and still have the same *behavior* (what the method does)
- See how your choice of implementing one method can affect the implementation of another.

## Background

Last week, you made two Block classes that have the same behavior. You can substitute one class for the other in any program that uses Blocks and nothing should change about how that program works. We continue looking at this important concept today by radically altering how your Deck class from Projects 2 works even though its behavior will be completely unchanged.

## Setup

Start a new project in Eclipse and add in three starter code classes from Nexus: "Client", "Deck" and "Card". Then look at the documentation for these classes (link also on Nexus). **Don't look at the code in Card.java**. Instead, try to understand the *behavior* of this Card class just through the Javadocs. Figure out how to construct new cards. Then practice a bit by going to the Client class in Eclipse and filling out the `sandbox` method that is just there for you to play around. Try to make a Queen of Diamonds. Use the other methods of Card. In a real software company, these Javadocs might be the only info you get about the Card class. You don't get the source. This is what we mean by behavior – you figure out *what* the class does through the docs even though you don't know *how* it works.

Now look at the Javadocs for Deck. It has a few more methods than yours did in Project 2. Read about each one in the docs (not just the summaries! Read the details too!) Then open up Deck.java in Eclipse. This is the class you are going to complete.

## Check out the Deck

The Deck class already has the constants and instance variables you need. *You are not allowed to change them or make any more of them.* While the constants should make sense, notice the instance variables. They are very different than your array-based Deck. The first is a `Vector`, an object that can store references to other objects. This structure will take the place of the array you used before. It's an example of an **Abstract Data Type (ADT)**, a data structure that can be defined solely by its behavior (i.e. its Javadocs). Go to the standard Java docs (link is on Nexus) and look up Vector. Like an array, it's an ordered list of items. Unlike an array, it has methods by which you can easily add items, remove items, and much more. You'll notice that we declare the variable in an odd way: `Vector<Card>`. This is because Vector is an example of a **parameterized class**. The parameter, `Card`, tells this ADT what kind of thing it will be holding. You'll work more with parameterized classes later in the course.

The second instance variable is the boolean `shuffled`. This tells if the deck is ordered (false) or randomized (true).

**IMPORTANT**: This lab is special because you'll be using the built-in Vector ADT. On all other labs and projects, you are not allowed to use Vector or any other built-in collection.

## The Details

Time to flesh out the Deck class. There are two big differences between this Deck's implementation and yours from Project 2. First, this Deck will always keep the cards in the vector in the same order. That is, once you insert the cards into the vector, you won't rearrange them. (So how do you shuffle? Keep reading…) Second, when cards are dealt, they will actually be removed from the vector. We do this since Vector gives us an easy way to remove cards, so we'll take advantage of it.

These two differences will greatly affect your code, but your job is still to get the same behavior out of this Deck that yours had in Project 2. Complete the following methods (in the following order):

1. Finish writing the constructor. I've already initialized the instance variables for you, but it's still unfinished. What's missing?

2. Write the `size` method. Remember, cards will be removed from the vector when they are dealt. **PITFALL ALERT**: Don't reinvent the wheel! Use the methods Vector provides.

3. Write the `isEmpty` method.

4. Write the `deal` method. This method either returns null if the deck is empty, removes and returns a random card if the deck is shuffled (the Random class will be helpful), or removes and returns the next card <u>in order</u> if the deck is unshuffled. Thus, it is this method that does the "shuffling" work by picking a card at random when it's time to deal a card from a shuffled deck!

5. Write the `shuffle` method. Be careful. Based on what `deal` is doing, what does this method have to do?

6. Write the `gather` method. Remember, the vector is probably not holding all the cards anymore, so you'll have to reinstantiate them. Don't construct a brand-new Deck object to do this. Note the postcondition in the comments of this method. A *postcondition* is something that should be true once this method finishes executing.

In the Client class, I have four methods commented out that test various methods in Deck. Uncomment them one at a time to test to see if your methods are working. Remember to read the docs to see what they are testing. You'll notice that each test also includes a commented call to `printStats`. If you'd like to see what the deck currently contains, uncomment this line.

## Don't miss the forest…

Once you are finished, take a moment to reflect on what you've done. You have just written a Deck of Cards that can be shuffled without shuffling. But the only person who has to know this is you -- the programmer. From the point of view of the person using your class for their own purposes, it works just like any other Deck of Cards. And that's the point. You can write or update existing code any way you want as long as the behavior is what the user expects to see. That might sound very freeing, but it is also a serious responsibility: you ***must*** maintain what the user expects to see, either because that's the way the object works in the real world or that's the way your Javadocs says it has worked in the past. If you fail to do this, you will, at best, confuse the user and, at worst, break other people's code which was built upon yours.

## How to turn in this lab

Before turning in any program in this class, remember our mantra: just because it works doesn't mean it's good! Keep your logic simple and straightforward. You still need meaningful names, neatness, good indentation, and whitespace to allow "breathing room" for the reader.

Please zip up your entire project (the folder named "<your name> Lab 4") and upload it using the link on Nexus. Don't forget to include a pdf file of your source code (just the classes you changed) and hand in a paper copy too.

Do not leave the lab until you've made a backup on orzo.