Lab 11 – A Touch of Class

1. Download and unzip **lab11_files.zip** from the class web page and open **BankAccount.py**, which contains the BankAccount definition we wrote in class. When loaded, this file creates three BankAccount objects called `b1`, `b2`, and `b3`. Test out these objects in the Python shell by interactively calling their methods `inquiry`, `withdraw`, and `deposit` a few times with different values.

2. Add the too-good-to-be-true method **restoreBalance(*self*)**, which resets the balance of an account back to its original opening amount. Hint: create a new instance variable called `self.originalBalance` to enable a BankAccount object to remember what its opening balance was at the time it was constructed. For example:

```
>>> b1.inquiry()
Balance is currently $35.00
>>> b1.restoreBalance()
Balance restored to $500.00
```

3. Now let's make BankAccount objects be password-protected. Modify the **__init__** constructor so that it takes an additional parameter *password* as input, which will be a password string. For example:

```
b1 = BankAccount(1001, 500.00, "open sesame")
```

   The account should process requests only if it is accompanied by the password with which the account was created, otherwise it should refuse the request. You'll need to add an extra *password* parameter to the other BankAccount methods, and modify their code accordingly. Be sure to test everything thoroughly. Here is a sample interaction:

```
>>> b1.withdraw(100, "abracadabra")
Sorry, that password is incorrect
>>> b1.withdraw(100, "open sesame")
Withdrew $100.00 from account #1001
>>> b1.inquiry("stick em up")
Sorry, that password is incorrect
>>> b1.inquiry("open sesame")
Balance is currently $400.00
```

4. Open the file **Coin.py**, which defines a simple Coin class. Running this code constructs a new Coin object with its `self.sideup` instance variable initialized to "Heads". Add a new method called **toss(*self*)** to the Coin class that simulates a single coin toss by randomly assigning a new value of "Heads" or "Tails" to `self.sideup` and printing out the result of the toss. For example:

```
>>> nickel.toss()
We got Tails
>>> print(nickel)
Tails
```

5. Next, add two new instance variables to Coin called `self.headCount` and `self.tailCount`, to keep track of the total number of times a Coin comes up Heads or Tails, respectively, when tossed. Also complete the method **reportStats(*self*)**, which should report the numbers of Heads and Tails obtained so far. You can test your code by calling the program testcoin(), which simulates ten coin tosses and reports the results.

6. Open the file **Student.py**, which contains the outline for a Student class, and examine the teststudent() program. Your job is to complete the class definition so that the test program works as shown below:

```
>>> teststudent()
Recorded a quiz score of 83 for Stu
Recorded a quiz score of 79 for Stu
Average quiz score for Stu is 81.0
```

(continued on next page)

```
Recorded a quiz score of 90 for Sue
Recorded a quiz score of 96 for Sue
Recorded a quiz score of 87 for Sue
Recorded a quiz score of 100 for Sue
Average quiz score for Sue is 93.25
```

A Student object should keep track of three instance variables: `self.name`, `self.scoreTotal`, and `self.count`. Calling the **record(*self, score*)** method should add the given quiz score to self.scoreTotal, increase self.count by 1, and print a message of the form "Recorded a quiz score of ___ for ___". The **reportAverage(*self*)** method should compute the student's current quiz average based on self.scoreTotal and self.count, and print a message of the form "Average quiz score for ___ is ___".

7. Open the file **Auditorium.py** and complete the Auditorium class definition for managing the number of open and filled seats in an auditorium. An Auditorium object has a seating capacity that is specified when it is constructed. Once this capacity is reached, no more seats can be filled. An Auditorium should keep track of its seating capacity, and the number of seats that are currently open. Write and test the following methods:

   - **seatsAvailable(*self*)** returns the current number of empty seats.
   - **seatsOccupied(*self*)** returns the current number of occupied seats.
   - **fillSeats(*self, numRequested*)** attempts to fill up to *numRequested* seats. If *numRequested* is greater than the number of currently available seats, all available seats are filled and a message reporting the number of requests that could be accommodated is printed.
   - **lookInside(*self*)** prints out the current number of filled and unfilled seats.

   Below are some examples of interacting with an Auditorium object:

```
>>> carnegieHall = Auditorium(1500)
>>> carnegieHall.lookInside()
No one is inside
>>> carnegieHall.fillSeats(1000)
Filled 1000 seats
>>> carnegieHall.lookInside()
1000 people inside with 500 seats left
>>> carnegieHall.fillSeats(700)
Sorry, sold out after filling 500 seats
>>> carnegieHall.lookInside()
1500 people inside with 0 seats left
>>> carnegieHall.fillSeats(100)
Sorry, all sold out
```

8. Open the file **PlayingCard.py**, which contains the outline of a PlayingCard class for representing individual playing cards, as well as another class called CardDeck for representing a full deck of 52 cards, and a test program war() that simulates the card game of War. Your job is to complete the PlayingCard class. The constructor should take two parameters: a number from 2 to 14 representing the card's rank (with 11=Jack, 12=Queen, 13=King, and 14=Ace), and a number from 1 to 4 representing the suit (1=Clubs, 2=Hearts, 3=Diamonds, 4=Spades). PlayingCard objects should have the following methods:

   - **getRank(*self*)** returns the rank as a number from 2 to 10 or the string "Jack", "Queen", "King", or "Ace"
   - **getSuit(*self*)** returns the suit as the string "Clubs", "Hearts", "Diamonds", or "Spades"
   - **__str__(*self*)** returns a string description of the card of the form "2 of Clubs", "Ace of Spades", etc.
   - **equals(*self, otherCard*)** returns True if this card has exactly the same rank and suit as *otherCard*
   - **trumps(*self, otherCard*)** returns True if this card has a higher rank than *otherCard*

   Test your class definition by creating some PlayingCard objects and calling their methods interactively at the Python prompt. When you are satisfied that your methods work, try running the test program war().