Lab Homework #2: Inheritance [Total: 55]
You must do this assignment individually and, unless otherwise specified, you must follow all the general instructions and regulations for assignments. Graders have the discretion to deduct up to 10% of the value of this assignment for deviations from the general instructions. Be sure to read them before starting.

Part 1: 0 points
Do not submit this part, as it will not be graded. However, doing these exercises might help you to do the second part of the assignment, which will be graded. If you have difficulties with the questions of Part 1, then we suggest that you consult the instructor during their office hours; they can help you and work with you through the warm-up questions.

Warm-up Question 1 (0 points)
Write a method printTypeOfChar. This method should take as input a char and print. If the char is upper case, your method should print UPPERCASE. If the char is lower case, your method should print LOWERCASE. If the char is any other symbol, your method should print SYMBOL. Note that it is not necessary to make 52 different cases if you consider the order of the Unicode chart and the fact that you can use the < and > operators on them.

Warm-up Question 2 (0 points)
Write a method countUppercase. Your method should take as input a String and return an int representing the number of upper case letters in the String. Now add a main method where you use the Scanner class to read a String from the user and call your method, outputting the returned result.

Warm-up Question 3 (0 points)
A prime number is a positive number whose only even divisors are 1 and itself. Write a method isPrime that takes as input an integer n and returns a boolean representing whether n is prime or not. Make sure to handle cases where the integer is negative. (Your method should return false in these cases.)

Warm-up Question 4 (0 points)
Write a method firstPrimeNumbers which takes as input an int n and returns an int[]. The int[] should contain the first n prime numbers.

Warm-up Question 5 (0 points)
x is a factor of y if y is a multiple of x. Write a method calculateFactors. The method should take as input an int n and return an int[] containing all the factors of the number n.

Part 2: Cards [30 points]
A standard deck of playing cards consists of 52 cards. Each card has a rank (2, 3, . . . , 9, 10, Jack, Queen, King, or Ace) and a suit (spades, hearts, clubs, or diamonds).

You will create a class called Card that will simulate cards from a standard deck of cards. Your class will extend the AbstractCard class (provided).

The ordering of the cards in a standard deck (as defined for this assignment) is first specified by the suit and then by rank if the suits are the same. The suits and ranks are ordered as follows:

suits: The suits will be ordered
diamonds < clubs < hearts < spades

ranks: The ranks will be ordered
2 < 3 < · · · < 9 < 10 < Jack < Queen < King < Ace

A Joker card is a special card that is "greater" than any other card in the deck (any two jokers are equal to each other). A joker has no suit (”None” from AbstractCard.SUITS).

Again, the overall ordering for non-joker cards is specified by suit first and then rank; for example, all club cards are "less than" all heart cards. Two cards with the same rank and suit are considered equal.

Write a Java class called Card that extends the provided Card class. Your class must have two constructors:

```
public Card(String rank, String suit)
{  // purpose: creates a card with given rank and suit
  // preconditions: suit must be a string found in Cards.SUITS
  // rank must be a string found in Cards.RANKS
  // Note: If the rank is AbstractCard.RANKS[15] then any
  // valid suit from AbstractCard.SUITS can be given
  // but the card’s suit will be set to AbstractCard.SUITS[4]
}

public Card(int rank, String suit)
  // purpose: creates a card with the given rank and suit
  // preconditions: suit must be a string found in Cards.SUITS
  // rank is an integer satisfying 1 <= rank <= 14, where
  // 1 for joker, 2 for 2, 3 for 3, .., 10 for 10
  // 11 for jack, 12 for queen, 13 for king, 14 for ace
  // Note: as with the other constructor, if a joker is created, any valid suit can be passed
  // but the card’s suit will be set to AbstractCard.SUITS[4]
}
```

Note that the case of strings is important here. The input strings must be exactly the same as those found in AbstractCard.SUITS or AbstractCard.RANKS.

The specification for the three abstract methods in the AbstractCard class are given by:

```
public int getRank()
{ // Purpose: Get the current card’s rank as an integer
  // Output: the rank of the card
```

```
  // joker -> 1, 2 -> 2, 3 -> 3, ..., 10 -> 10
  // jack -> 11, queen -> 12, king -> 13, ace -> 14
}

public String getRankString()
{ // Purpose: Get the current card's rank as a string
  // Returns the cards's rank as one of the strings in Card.RANKS
  // (whichever corresponds to the card)
}

public String getSuit()
{ // Purpose: Get the current card's suit
  // Returns the card's suit as one of the strings in Card.SUITS
  // (whichever corresponds to the card)
}
```

Do not change the provided AbstractCard class. You can add any (non-static) attributes and helper methods that you need for your Card class. By using your completed code make screenshot demonstrating the following sample input/output.

Sample Input/Output:

```
Card c = new Card("Queen", "Diamonds");
c.getRank();          returns      12
c.getRankString();    returns      "Queen"
c.getSuit();          returns      "Diamonds"
System.out.println(c); displays    12D
Card d = new Card("4", "Spades");
c.compareTo(d);       evaluates to some negative int
d.compareTo(c);       evaluates to some positive int
Card e = new Card("Jack", "Spades");
d.compareTo(e);       evaluates to some negative int
e.compareTo(e);       evaluates to 0
e.getRank();          evaluates to 11
e.getSuit();          evaluates to "Spades"
Card j = new Card(1, "None");
System.out.println(j); displays    "J"
j.getRankString();    returns      "Joker"
j.getRank();          returns      1
j.getSuit();          returns      "None"
e.compareTo(j);       evaluates to some negative integer
```

Part 3: JUnit Test of the Card Class [25 points]

Earlier, we have demonstrated how to implement JUnit test cases in order to ensure the validity of the program. In this part, the students should use the attached JUnit test class and implement the missing test cases provided in that class. [Add JUnit 4 library in your project]

There are two example test cases provided to the students so that they can follow similar strategy to create the remaining tests. In comments the purpose of the test has been explained for each test cases. One example test cases is shown in the following:

```
@Test
public void testRank() {
        // This test evaluates whether the
        // getRank function has been implemented correctly

        Card c = new Card("Queen", "Diamonds");
        int actual = c.getRank();
        int expected = 12;

        assertEquals(expected, actual);
}
```

The assignment rubrics for the assignment is given in the following:

Criterion	Details	Deductions
Classes	At minimum, we need the Cards.java, CardTest.java files	-7: constructor implementations are incomplete -7: getRank function is incomplete -7: getSuit function is incomplete -7: compareTo implementation is incomplete -3 for each missing test or incomplete test
Code quality	Identifier names, class names, proper use of public/private, ample comments in main, etc.	-15: incoherent, inconsistent coding style -10: comments were not used throughout the code
Test	Note, whether the program compile and run	-55: The portion of the code of the assignment is a copy -55: The required java files not submitted -55: The program does not compile -20: screenshot of the program run is not attached -20 screenshot of the JUnit test is not attached