COMP1006/1406 – Winter 2022

Submit a single zip file called `a4p1.zip` to cuLearn.

There is NO coding in this assignment. It is meant to be done entirely by hand.
As with the previous assignments, there is a 48-hour grace period in which assignments will be accepted without penalty.

This part of the assignment has 10 marks.

1 Testing [3 marks]

Recall the `SpecificBox` class from Assignment 2 - Part 2. When testing your class, what is the *minimum* number of test cases that should be used? Provide these (minimum number of) test cases in a text file called `tests.txt`. Each test case in the file should have the following format (for example, if this is your 3rd test)

```
// Test 3
// TESTING - testing XYZ
box1 = new SpecificBox(label1, location1, size1);
box2 = new SpecificBox(label2, location2, size2);
expected = exp;                    // an integer
actual = box1.comppareTo(box2);
```

Here, label1, label2, location1, location2, size1, size2, and exp are values you supply. The TESTING comment should provide a *brief* description of what is being tested. Note that pasting any of your test cases in to the following code should work:

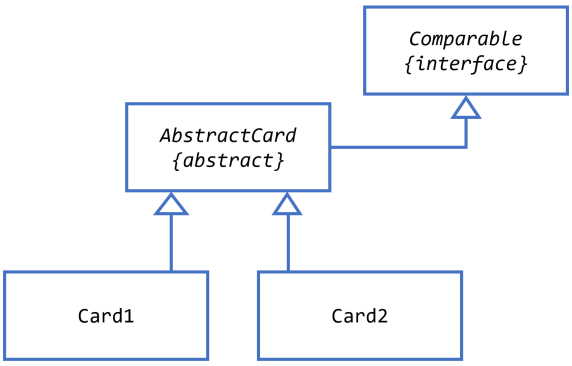
```
SpecificBox box1, box2;
int expected, actual;
//
// your test copied here
//
System.out.println("test passed : " + (expected == actual);
```

At the end of your `tests.txt` file, *briefly* explain what you need these number of tests.

2 ♠♥♣♦ Cards ♦♣♥♠ [7 marks]

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 ♦).

Suppose that there is an **abstract** class called `AbstractCard` that implements the interface `Comparable<AbstractCard>`, but does not override the `compareTo()` method. Suppose also that `AbstractCard` has two direct sub-children: `Card1` and `Card2`. The class hierarchy is as follows:



The `AbstractCard` class has a constructor that takes a `String` as input and sets the state of the card (the rank and suit) as follows:

```
public AbstractCard(String card)
  // purpose: sets the state of a card based on the input string
  // preconditions: card is a short textual representation of a card
  //                it always has rank (2,3,...,9,10,J,K,Q,A) that is
  //                followed by suit (S,D,C,H) in upper-case
  // postconditions: sets the rank and suit state of the object
  // examples: Card("2S")  -> two of spades
  //           Card("3D")  -> three of diamonds
  //           Card("9C")  -> nine of clubs
  //           Card("10H") -> 10 of hearts
  //           Card("JS")  -> jack of spades
  //           Card("KD")  -> king of diamonds
  //           Card("QC")  -> queen of clubs
  //           Card("AH")  -> ace of hearts
```

Both subclasses (`Card1` and `Card2`) will have a constructor that takes a string (same format as `AbstractCard`, called `card`) and calls `super(card)` when creating an object.

The `AbstractCard` class also overrides the `toString()` method to return a string representation of a playing card that follows the same form as the inputs to the constructor. So, cards are displayed (when printed) like `3H`, `AC`, `10D`, etc. It is overridden as a `final` method. Note that we will refer to individual cards (objects) using this representation when convenient.

Card1 In the `Card1` class, the `compareTo()` method orders cards by first comparing the card's suits and then ranks if needed (when there is a tie). The suits and ranks are ordered as follows:

suits: The suits will be ordered

$$\text{diamonds } ♦ < \text{clubs } ♣ < \text{hearts } ♥ < \text{spades } ♠$$

ranks: The ranks will be ordered (READ CAREFULLY)

$$2 < 3 < \cdots < 9 < 10 < \text{Jack} < \text{King} < \text{Queen} < \text{Ace}$$

Here are some examples,

```
Card1 queen_of_hearts = new Card1("QH");
Card1 queen_of_clubs = new Card1("QC");
Card1 ten_of_spades = new Card1("10S");
Card1 seven_of_spades = new Card1("7S");
// assert: queen_of_hearts.compareTo(queen_of_clubs)  > 0
// assert: queen_of_hearts.compareTo(seven_of_spades) < 0
// assert: ten_of_spades.compareTo(seven_of_spades)   > 0
```

Card2 In the `Card2` class, the `compareTo()` method orders cards solely by rank as follows:

ranks: The ranks will be ordered (READ CAREFULLY)

$$\text{Ace} < 2 < 3 < \cdots < 9 < 10 < \underbrace{\text{Jack} = \text{King} = \text{Queen}}_{\text{considered equal}}$$

Here are some examples,

```
Card2 queen_of_hearts = new Card1("QH");
Card2 queen_of_clubs = new Card1("QC");
Card2 ten_of_spades = new Card1("10S");
Card2 seven_of_spades = new Card1("7S");
// assert: queen_of_hearts.compareTo(queen_of_clubs)  = 0
// assert: queen_of_hearts.compareTo(seven_of_spades) > 0
// assert: seven_of_spades.compareTo(ten_of_spades)   < 0
```

Now suppose that you have an array (`AbstractCard[]`) that contains both `Card1` objects and `Card2` objects and you want to sort the array using bubble sort. The pseudocode for bubble sort is as follows:

```
procedure bubbleSort(A : list of sortable items)
   n := length(A)
   print "initial:", the list A
   repeat
       swapped := false
       for i := 1 to n-1 inclusive do
           /* if this pair is out of order */
           if A[i-1] > A[i] then                          // <- this is the red line
               /* swap them and remember something changed */
               swap(A[i-1], A[i])
               swapped := true
           end if
       end for
       n := n - 1
       print "it {i}", the list A
   until not swapped
end procedure
```

The line in red (above) is problematic for a couple of reasons. One issue is that this line can translate into java in two ways:

```
version 1:    if( A[i-1].compareTo(A[i]) > 0 ){
version 2:    if( A[i].compareTo(A[i-1]) < 0 ){
```

If we run the bubble sort algorithm using version 1 of the red line on the following list

```
AbstractCard[] cards = {new Card1("QD"),new Card1("9H"),new Card1("JD"),new Card1("AD")};
```

the output (what is printed) would be as follows: (colours will NOT be shown, the red indicate values that are fixed at the end of the iteration of the repeat loop)

```
initial: [QD, 9H, JD, AD]
   it 1: [QD, JD, AD, 9H]
   it 2: [JD, QD, AD, 9H]
   it 3: [JD, QD, AD, 9H]
```

For each of the following cases, run the bubble sort algorithm as described in this question. **Display the output of each print statement in the bubble sort algorithm as done above**. Add appropriate padding so that the colons are lined up in the output (as above). You do NOT have to use any colours for this.

(A) Run the bubble sort pseudocode (**using version 1** of the red line) on the following list

```
AbstractCard[] cards = {new Card2("QD"),new Card2("9H"),new Card2("JD"),new Card2("AD")};
```

(B) Run the bubble sort pseudocode (**using version 1** of the red line) on the following list

```
AbstractCard[] cards = {new Card2("QD"),new Card1("9H"),new Card1("JD"),new Card2("AD")};
```

(C) Run the bubble sort pseudocode (**using version 2** of the red line) on the following list

```
AbstractCard[] cards = {new Card2("QD"),new Card1("9H"),new Card1("JD"),new Card2("AD")};
```

In addition to the code traces, describe the results. Why are the 'sorted' lists different? Explain why care must be taken when different subclasses override the `compareTo()` method differently. Write up your solution to this question in a text file called `sorting.txt`.

Submission Recap

Submit ONE zip file called `a4p1.zip` that has TWO text files in it: `tests.txt` and `sorting.txt`.