COMP1006/1406 – Fall 2023

Submit all three of your assignment files to gradescope.
This assignment has 10 marks.

Part One: Your code is marked by the autograder in gradescope. It is all based on correctness.

Part Two: Your plain text file will be hand marked.

Part Three: Your pdf will be hand marked.

Part One

Comparable [4 marks]

Consider the following abstract class.

```
public abstract class Box implements Comparable<Box>{
   String label;
   String location;
   int    size;
   public String getLabel(){ return this.label;}
   public int getSize(){ return this.size;}
   public String getLocation(){ return this.location;}
}
```

You will complete the provided `SpecificBox` class which extends the `Box` class. The `SpecificBox` class must be a **concrete** class. The `SpecificBox` class must be implemented so that a total ordering is imposed on `SpecificBox` objects as follows: Let X and Y be `SpecificBox` objects, then (in this order)

a) $X < Y$ whenever X's location is alphabetically *less than* Y's location. So, `X.compareTo(Y)` will be negative and `Y.compareTo(X)` will be positive, or

b) $X < Y$ whenever X and Y have the same location and the length of X's label $<$ the length of Y's label, or

c) $X < Y$ whenever X and Y have the same location, the lengths of their labels are the same, and X's size is **larger** than Y's size, or

d) $X = Y$ whenever X and Y have the same location, their label lengths are the same and their sizes are the same.

e) Otherwise, $Y < X$

Another way of specifying the ordering would be to consider a sorted list of `SpecificBox` objects. They would first be sorted by location (alphabetically[1]), then for ties (that is, having the same location) sorted by length of labels, and finally for ties in both sorted by size from largest to smallest.

[1] Using the natural ordering imposed by the `String` class.

Java's `Comparable` interface does not specify *which* negative/positive integers the `compareTo()` method should return (any will do!). However, in this assignment, when you implement your `SpecificBox` class, the `compareTo()` method **must** return one only of seven different numbers as follows: If the locations of the boxes are different it should return ±1, if the locations are the same, but the lengths of the labels are different it should return ±2, if the locations and label lengths are the same, but the size is different it should return ±3. Finally, if they are the same box (by state) then it should return 0 (zero).

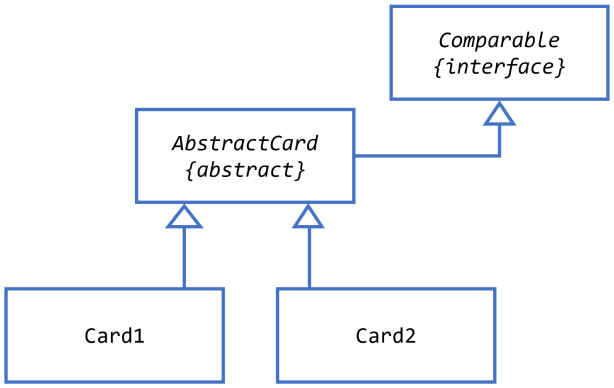
Submit your `SpecificBox.java` file to gradescope.

Part Two

♠♥♣♦ Cards ♦♣♥♠ [4 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.

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

$$\text{diamonds } \blacklozenge < \text{clubs } \clubsuit < \text{hearts } \heartsuit < \text{spades } \spadesuit$$

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
```

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 yellow 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 yellow (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 yellow 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 **by hand** as described in this question and **display the output of <u>each</u> print statement in the bubble sort algorithm as done above**. Add appropriate spacing (spaces not tabs) so that the colons and cards are lined up in the output (as above).

(A) Run the bubble sort pseudocode (**using version 1** of the yellow 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 yellow 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 yellow 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, part **(D)** of this problem is to describe the results. Why are the 'sorted' lists different? Explain why care must be taken when different subclasses override the `compareTo()` method differently. This problem is to be submitted in a plaintext file called `sorting.txt`. Be sure your submit a plaintext file (not an `.rtf` file, a `.docx` file, a `.pdf` file, etc.). You can use VS Code as a text editor for this.

Submit your `sorting.txt` to gradescope.

Part Three

Drawing **[2 marks]**

We have reached the half-way point (more or less) in the semester. Think about your experience so far in COMP 1006/1406. Think about what you have learned and what you have done. The joys and frustrations. Think about what you might be able to do with what you have learned. Your task in this problem is to either draw a picture that expresses this reflection or to write about it (or a combination of both). My hope in asking you to do this exercise is that you will critically reflect on what you have learned and perhaps where you would like to take what you have learned forward. It should also make this assignment a bit lighter than the others. The intention is that this problem should not cause you any stress. Do not worry about your "artistic ability". You will not be graded on how "artistic" your drawing is or how grammatically correct your writing is. If you put an honest effort into the problem, you will receive full marks. Have fun!

You can create your drawing or writing in any way you wish but you should save it in PDF format. Ideally, the size of your drawing should be standard letter-size in horizontal orientation and the length of writing should not be more than one page.

If you want your submission to remain private (and not shown to the class), save your file as `private-name.pdf`, where `name` is your name. If you agree to have your picture/text possibly displayed (this semester or in future semesters of this course or related courses), submit your drawing in a file called `public-name.pdf`, where `name` is your first (given) name. For public submissions, TAs may post their *facvourites* to dicord.

Submit your pdf file to gradescope.

Note: For public submissions, do NOT include your full name/ID in your picture/text unless you are OK with everyone seeing it. Since you are submitting using Brightspace, we already know who you are so we don't need this information in your picture.

Note: Offensive/rude/insensitive submissions will receive zero marks and may be forwarded to the Dean's office depending on the severity. (This has never happened before, and I do not anticipate it happening now.)

Submission Recap

A full submission consists of three files: `SpecificBox.java`, `sorting.txt` and a `pdf` file (with appropriate name).

Each time you subnmit, be sure to include **all** files you have ready to submit to ensure that all files are considerewd when grading.
Note that `SpecificBox.java` will be graded for correctness by the autograder. The remaining questions will be graded by hand.