Lecture 9

Sorting in Java
Comparable and Comparator

Last time:

We looked at three simple sorting algorithms that run in O(n^2) time in the worst-case:

- Bubble Sort: Per round, ***BUBBLES*** *up the biggest value* to the top (or to the right). It is very simple but very slow. Each round, the right-hand side of the array is being sorted.
- Selection Sort: ***SELECTS*** *the smallest value* and swap it with the left most value of an array (Left side increases by one per round). On the contrary to the bubble sort, per round, the left-hand side of the array is being sorted. It is faster than the bubble sort due to the fact that there is less number of swaps.
- Insertion Sort: Using an imaginary dividing line, ***INSERTS*** *the first value* of the right-hand side of the line into the proper position on the left-hand side of the line. In most cases, it is faster than the bubble sort and the selection sort because it requires less number of comparisons on average and uses copies instead of swaps. In fact, its best-case running time complexity is O(n).

Now, how does sorting in Java work?

Primitives

We can simply sort an array of primitives by invoking Arrays.sort method.

Objects

To sort an array or a collection of objects, we need to make sure that the objects are mutually **comparable**.

This is supported by the Comparable interface that contains only one method:

```
compareTo(T) : int
```

The return value of the compareTo() method is negative, zero or positive:

- Negative when the current instance comes before the argument in the ordering
- Positive when the current instance comes after the argument.
- Otherwise zero is the return value

Let's take a look at the following code example.

```
public class Card _____________________________________________ {

    private String suit;
    private int rank;

    public Card(String s, int r) {
        suit = s;
        rank = r;
    }

    public String getSuit() {
        return suit;
    }

    public int getRank() {
        return rank;
    }
    /**
     * returns negative if this card's rank < other’s rank
     * returns 0 if this card's rank == other’s rank
     * returns positive if this card’s rank > other’s rank
     */
    @Override
    public int compareTo(Card other) {
        return ______________________________________________;
    }

    @Override
    public String toString() {
        return suit + ", " + rank;
    }
}
```

It is very important to know that if a class implements the Comparable interface, then the class’s `compareTo()` should be consistent to `equals()` such that if `x.equals(y) == true`, then `x.compareTo(y) == 0`.

Keep in mind that the default `equals()` method compares two objects using their references (identity).

Quick quiz!

```
Card card1 = new Card(“hearts”, 1);
Card card2 = new Card(“diamonds”, 1);
```

With the two cards above, what would be the output of the following code?

```
System.out.println(card1.compareTo(card2));
```

What about the following code?

```
System.out.println(card1.equals(card2));
```

Also, don’t forget that you must implement `hashCode()` method too! *If* `x.equals(y)` *is true, then* `x.hashCode()` *must be same as* `y.hashCode()`*.*

Note

Objects that implement the Comparable interface can be sorted by the `sort()` method of the Arrays or Collections class.

Question 1: what would be the output of the following code?

```
List<Card> cards = new ArrayList<Card>();

cards.add(new Card("hearts", 2));
cards.add(new Card("diamonds", 2));
cards.add(new Card("spades", 3));
cards.add(new Card("clubs", 4));
cards.add(new Card("hearts", 1));

_____________________.sort(cards);

for (Card c : cards) {
    System.out.println(c);
}
```

What if we want to have a few different ways of comparing cards?

For example, comparing cards only by suit. The problem is that there is **only one `compareTo()` method** by implementing the Comparable interface.

Luckily, Java provides another interface, Comparator.

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

The compare method returns a negative integer, zero, or a positive int value as the first argument is less than, equal to, or greater than the second.

Pay attention to the fact that you need another class that implements the Comparator interface and `compare()` method takes two arguments!

```
import java.util.*;
public class CompareBySuit implements Comparator<Card> {
    @Override
    public int compare(Card x, Card y) {
        return _____________________________________________;
    }
}
```

Question 2: Now what would be the output of the following code?

```
List<Card> cards = new ArrayList<Card>();

cards.add(new Card("hearts", 2));
cards.add(new Card("diamonds", 2));
cards.add(new Card("spades", 3));
cards.add(new Card("clubs", 4));
cards.add(new Card("hearts", 1));

__________________.sort(cards, ________________________________);

for (Card c : cards) {
    System.out.println(c);
}
```

Notice that it is a separate class that implements the Comparator interface. This can be implemented as a nested class.

I want to also sort Cards using both suit and rank, which means I want to compare cards first by suit and, if they are the same, then compare them by rank.

```
import java.util.*;
public class CompareBySuitRank implements Comparator<Card> {
    @Override
    public int compare(Card x, Card y) {
        int suitResult = ____________________________________;
        if (suitResult != ___________) {
            return ____________________________________________;
        }
        return ________________________________________________;
    }
}
```

Question 3: Now, what would be the output of the following code?

```
List<Card> cards = new ArrayList<Card>();

cards.add(new Card("hearts", 2));
cards.add(new Card("diamonds", 2));
cards.add(new Card("spades", 3));
cards.add(new Card("clubs", 4));
cards.add(new Card("hearts", 1));

_________________.sort(cards, ________________________________);

for (Card c : cards) {
    System.out.println(c);
}
```

Tip!
compareTo() method defines the NATURAL ORDER of the type whereas compare() method is used to define different custom orders of the type.