

Lecture 9

Sorting in Java

Comparable and Comparator

Last time:

We looked at three simple sorting algorithms that run in $O(N^2)$ time:

- Bubble Sort: Per round, **BUBBLES** up the *biggest value* to the top (or right). It is very simple but very slow. Each round, right side of the array is 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). Per round, the left side of an array is sorted. It could be faster for a small amount of data than the bubble sort due to the fact that there are less swaps.
- Insertion Sort: Using an imaginary division line, **INSERTS** the *first value* of the right-hand side of the line into the proper position in the left-hand side of the line. In most cases, it is faster than the bubble sort and the selection sort because it requires smaller number of comparisons and no swaps but copies.

Now, how does sorting work in Java?

Primitives

We can simply sort an array of primitives by direct invocation of `Arrays.sort` method.

Objects

To sort 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(element: Object) : 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 implements Comparable<Card> {
    private String suit;
    private int rank;

    public Card(String suit, int rank) {
        this.suit = suit;
        this.rank = rank;
    }

    public String getSuit() {
        return suit;
    }

    public int getRank() {
        return rank;
    }
}
```

```
/**
 * returns negative if this card's rank is smaller
 * returns equals if this card's rank is equal
 * returns positive if this card's rank is
 */
@Override
public int compareTo(Card x) {
    return getRank() - x.getRank();
}

@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.compareTo(y) == 0`, then `x.equals(y) == true`.

The default `equals()` method compares two objects using their reference numbers.

Quick quiz!

```
Card card1 = new Card("heart", 1);
Card card2 = new Card("diamond", 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 `hashCode()` method!

If `x.equals(y)` is true, then `x.hashCode() == y.hashCode()`.

Note

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

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

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

Card card1 = new Card("heart", 2);
Card card2 = new Card("diamond", 2);
Card card3 = new Card("spade", 3);
Card card4 = new Card("club", 4);
Card card5 = new Card("heart", 1);

cards.add(card1); cards.add(card2); cards.add(card3);
cards.add(card4); cards.add(card5);

Collections.sort(cards);

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

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

For example, compare cards 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 integer 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 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>();

Card card1 = new Card("heart", 2);
Card card2 = new Card("diamond", 2);
Card card3 = new Card("spade", 3);
Card card4 = new Card("club", 4);
Card card5 = new Card("heart", 1);

cards.add(card1); cards.add(card2); cards.add(card3);
cards.add(card4); cards.add(card5);

Collections.sort(cards, new CompareBySuit());

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

Again, notice that it is a separate class that implements the Comparator interface.

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 _____;
        else
            return _____;
    }
}
```

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

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

Card card1 = new Card("heart", 2);
Card card2 = new Card("diamond", 2);
Card card3 = new Card("spade", 3);
Card card4 = new Card("club", 4);
Card card5 = new Card("heart", 1);

cards.add(card1); cards.add(card2); cards.add(card3);
cards.add(card4); cards.add(card5);

Collections.sort(cards, new CompareBySuitRank());

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