Introduction to Searching and Sorting

- Comparable Interface
- Comparator Interface
- Algorithm Complexity Classes
- Exercises
The Comparable Interface

- The Comparable interface of java.lang

```java
public interface Comparable{
    public abstract int compareTo(Object object);
}
```

- If a class implements Comparable interface, the compareTo method defines the natural ordering of its objects when sorted.

- By repeated calls to compareTo method, The sorting algorithms will be able to sort, according to the natural ordering, a list of objects belonging to a class implementing Comparable interface.

- Several core Java classes implement Comparable.

- A user defined class that implements Comparable should implement the compareTo method such that: object1.compareTo(object2) is:

  0 if object1 “is equal to” object2

  > 0 if object1 “is greater than” object2

  < 0 if object1 “is less than” object2
The Comparable Interface (cont’d)

- It is also preferable for `object1.compareTo(object2)` to return 0 if and only if `object1.equals(object2)` is true.

- The `compareTo` method throws a `ClassCastException` if the type of this object and the type of the object passed as parameter are not compatible for comparison.

- **Example 1:** A `BankAccount` defining the natural ordering as the ascending order of accountNumbers.

```java
import java.util.*;

class BankAccount implements Comparable{
    private int accountNumber;
    private String name;
    private double balance;
    public int compareTo(Object object){
        BankAccount account = (BankAccount) object;
        if(accountNumber < account.accountNumber)
            return -1;
        else if(accountNumber == account.accountNumber)
            return 0;
        else
            return 1;
    }
    // Other methods...
}
```
The Comparable Interface (cont’d)

```java
public String toString()
{
    return "Account#: " + accountNumber + " , Name: " + name + " , Balance: " + balance + " SR";
}
```

Example:

Assuming that `account1` and `account2` are `BankAccount` objects, a typical call to the `compareTo` method is:

1. int comparisonResult = account1.compareTo(account2);
2. if(comparisonResult == 0)
3.    System.out.println("Same account");
4. else
5.    System.out.println("Different accounts");

- If we want to sort objects of a class which does not implement Comparable interface, or the class implements Comparable but we want To order its objects in a way different from the natural ordering defined by Comparable, the `java.util.Comparator` interface should be used.
The Comparator Interface

- The Comparator interface is one of the *java collections framework* interfaces.
- The *Java collection framework* is a set of important utility classes and interfaces in the *java.util* package for working with collections.
- A *collection* is a group of objects.
- *Comparator* interface defines how collection objects are compared.

```java
public interface Comparator {
    public abstract int compare(Object object1, Object object2);
    public abstract boolean equals(Object object);
}
```

A class that implements Comparator should implement the `compare` method such that its return value is:

- 0 if `object1` “is equal to” `object2`
- > 0 if `object1` “is greater than” `object2`
- < 0 if `object1` “is less than” `object2`
The Comparator Interface (cont’d)

• It is also preferable for the compare method to return 0 if and only if object1.equals(object2) is true.

• The compare method throws a ClassCastException if the type of object1 and that of object2 are not compatible for comparison.

• The equals method concerns the Comparator object. It returns true if its parameter is a Comparator object and if it uses the same ordering as the invoking (calling) Comparator object; otherwise it returns false.

• Note: Since each class inherits the equals method from the Object class, it is not necessary for a class that implements the Comparator interface to implement the equals method.

• In our examples concerning the Comparator interface, the equals method will not be implemented.
Example 2: This example sorts the strings in reverse order of the alphabetical one.

```java
import java.util.*;

class StringReverseComparator implements Comparator {
    public int compare(Object object1, Object object2) {
        String string1 = object1.toString();
        String string2 = object2.toString();
        // Reverse the comparison
        return string2.compareTo(string1);
    }
}

class Test {
    public static void main(String[] args) {
        Arrays.sort(array, new StringReverseComparator());
        System.out.println(Arrays.asList(array));
    }
}
```

- The sort method, in the Arrays class, sorts the array “array” according to the comparator object. Notice the comparator object provided as a parameter for the sorting method.
- After printing, we get the following order:
  [Omar, Mohammad, Hisham, Hassan, Bilal, Ali, Ahmad]
The Comparator Interface (cont’d)

Example 3: Here the order of the comparator is the descending order of the absolute values

```java
import java.util.*;

class yComparator implements Comparator {
    public int compare(Object obj1, Object obj2) {
        int i1 = ((Integer)obj1).intValue();
        int i2 = ((Integer)obj2).intValue();
        return Math.abs(i2) - Math.abs(i1);
    }
}

class TestCollections2 {
    public static void main(String args[]) {
        Integer[] array = {new Integer(-200), new Integer(100),
                           new Integer(400), new Integer(-300)};
        Arrays.sort(array);
        System.out.println("Natural ordering: "+ Arrays.asList(array));
        Arrays.sort(array, new yComparator());
        System.out.println("My own ordering : "+ Arrays.asList(array));
    }
}
```
The Comparator Interface (cont’d)

Example 4: Here we want to order the BankAccount objects, not according to their natural ordering as defined before, but according to the opposite of the alphabetical order of the names of their owners.

```java
2  class MyComparator implements Comparator{
3  public int compare(Object object1, Object object2){
4      BankAccount account1 = (BankAccount) object1;
5      BankAccount account2 = (BankAccount)object2;
6      String string1 = account1.getName();
7      String string2 = account2.getName();
8      // Reverse the comparison
9      return string2.compareTo(string1);
10  }
11 }
```
Algorithm Complexity Classes

• Different algorithms require different amount of running time and space.
• The less amount of running time the more time-efficient the algorithm.
• The less amount of space requirements the more space-efficient the algorithm.
• The resources (such as time and space) required to solve a problem usually increase with an increase in the size n of the problem.
• Several factors affect the time and space requirements of an algorithm: hardware, language of implementation, Compiler being used, etc.
• The average running time of an algorithm is a function f(n) of the problem size n.
• Algorithms are classified into different groups (called complexity classes) based on f(n)
### Algorithm Complexity Classes (cont’d)

<table>
<thead>
<tr>
<th>Algorithm type</th>
<th>Average running-time is proportional to:</th>
<th>more time-efficient</th>
<th>less time-efficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant-time</td>
<td>independent of n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logarithmic-time</td>
<td>log n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linear-time</td>
<td>n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n log n time</td>
<td>n log n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quadratic-time</td>
<td>n(^2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cubic-time</td>
<td>n(^3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polynomial-time</td>
<td>n(^k) (k is a constant)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exponential-time</td>
<td>k(^n) (k is a constant)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercises

1. Write a Comparator to compare names by last name.

2. Write a Comparator to compare names by middle name. Assume that each name to be compared, consists of three names: first, middle, and last.

3. Arrange algorithm complexity classes from the most time-efficient to the least time-efficient.