Comparing Objects

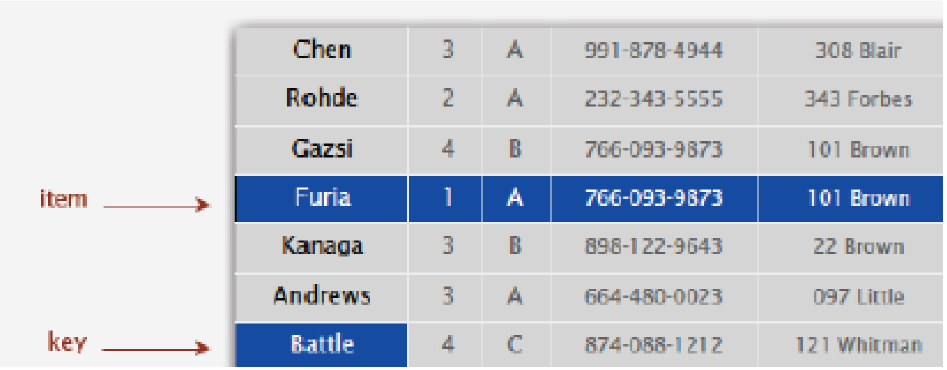
Frank Walsh

Goal of sorting

- The objective of a sorting algorithm is to rearrange the items such that their keys are ordered according to some well-defined ordering rule (usually numerical or alphabetical order)
- Each item contains a key
- Keys as SORTABLE.

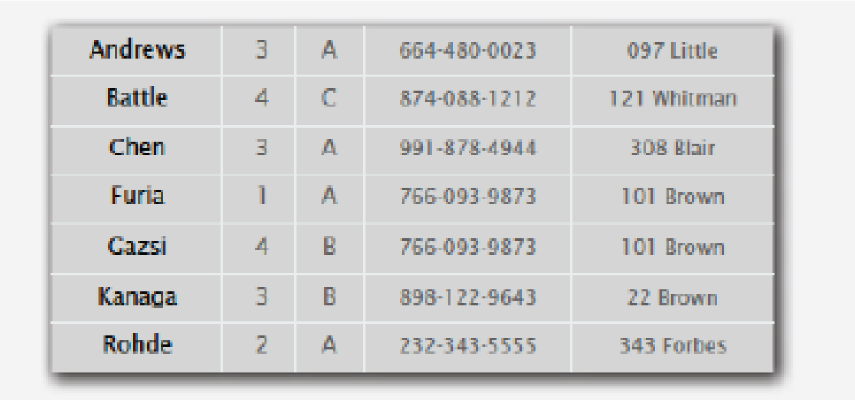
Example

- Unsorted



Chen	3	A	991-878-4944	308 Blair
Rohde	2	A	232-343-5555	343 Forbes
Gazsi	4	B	766-093-9873	101 Brown
Furia	1	A	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman

- Sorted



Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

Two Sorts

```
// Create a list of strings
    ArrayList<String> al = new ArrayList<String>();
    al.add("Geeks For Geeks");
    al.add("Friends");
    al.add("Dear");
    al.add("Is");
    al.add("Superb");

    /* Collections.sort method is sorting the
    elements of ArrayList in ascending order. */
    Collections.sort(al);

    // Let us print the sorted list
    System.out.println("List after the use of" +
            " Collection.sort() :\n" + al);
```

```
// Create a list of Users
    ArrayList<String> al = new ArrayList<String>();
    al.add(new User(“Frank”,”Walsh”));
    al.add(new User(“Mary”,”Power”));
    al.add(“new User(“Frank”,”Dawson”));
    al.add(new User(“Jack”,”OConor”));
    al.add(new User(“Bob”,”Dylan”));

    /* Collections.sort method is sorting the
    elements of ArrayList in ascending order. */
    Collections.sort(al);

    // Let us print the sorted list
    System.out.println("List after the use of" +
            " Collection.sort() :\n" + al);
```

Comparable Interface

- Q.How does the same sort() method in previous examples work with Files, Strings, Doubles???
- A.They all implement the Comparable interface. (Remember interfaces from 1st Week)
- Sometimes known as “Callback”

Comparing Stuff

- Four methods underlie many of Java’s important Collection types: equals, compare and compareTo, and hashCode
  - To put your own objects into a Collection, you need to ensure that these methods are defined properly
  - Any collection with some sort of *membership test* uses equals (which, in many cases, defaults to ==)
  - Any collection that depends on *sorting* requires larger/equal/smaller comparisons (compare or compareTo)
  - Any collection that depends on *hashing* requires both equality testing and hash codes (equals and hashCode)
  - Any time you implement hashCode, you *must* also implement equals
- Some of Java’s classes, such as String, already define all of these properly for you
  - For your own objects, you have to do it yourself

Comparing our own objects

- The Object class provides public boolean equals(Object obj) and public int hashCode() methods
  - For objects that we define, the inherited equals and hashCode methods use the object’s address in memory
  - We can override these methods
  - If we override equals, we *should* override hashCode
  - If we override hashCode, we *must* override equals
- The Object class does not provide any methods for “less” or “greater”—however,
  - There is a Comparable interface in java.lang
  - There is a Comparator interface in java.util

Outline of a **Student** class

```java
public class Student implements Comparable<Student> {

public String name;
public int score;

public Student(String name, int score) {
        this.name = name;
        this.score = score;
        }

@Override
        public int compareTo(Student that) {
                return this.score-that.score;
}
}
```

Include a main method

```
public static void main(String args[]) {
      TreeSet<Student> set = new TreeSet<Student>();
      set.add(new Student("Ann", 87));
      set.add(new Student("Bob", 83));
      set.add(new Student("Cat", 99));
      set.add(new Student("Dan", 25));
      set.add(new Student("Eve", 76));
      Iterator<Student> iter = set.iterator();
      while (iter.hasNext()) {
          Student s = iter.next();
          System.out.println(s.name + "  " + s.score);
      }
}
```

Using the TreeSet

- Use an iterator to print out the values in order, and get the following result:

```
Dan  25
Eve  76
Bob  83
Ann  87
Cat  99
```

- Iterator "knows" that it should sort Students by score, rather than, say, by name from the compareTo() method.

Using a separate Comparator

- In the program we just finished, Student implemented Comparable
  - Therefore, it had a compareTo method
  - We could sort students *only* by their score
  - If we wanted to sort students another way, such as by name, we are out of luck
- Now we will put the comparison method in a *separate class* that implements Comparator instead of Comparable
  - This is more flexible (you can use a different Comparator to sort Students by name or by score), but it's also clumsier
  - Comparator is in java.util, not java.lang
  - Comparable requires a definition of compareTo but Comparator requires a definition of compare

Outline of StudentComparator

```
public class StudentComparator implements
Comparator<Student> {


@Override
public int compare(Student s1, Student s2) {
......
}


}
```

- Note: When we are using this Comparator, we don't need the compareTo method in the Student class

The compare method

```
public int compare(Student  s1, Student s2) {
        return s1.score – s2.score;
}
```

- This differs from compareTo(Object o) in Comparable in these ways:
    - The name is different
    - It takes both objects as parameters, not just one

Update **main** method

- The main method is just like before, except that instead of

```
TreeSet<Student> set = new TreeSet<Student>();
```

  We have

```
Comparator<Student> comp = new StudentComparator();
TreeSet<Student> set = new TreeSet<Student>(comp);
```

When to use each

- The Comparable interface is simpler and less work
  - Your class implements Comparable
  - You provide a public int compareTo(…) method
  - You will use the same comparison method every time
  - Use for “natural” or “default” sort order.
- The Comparator interface is more flexible but slightly more work
  - Create as many different classes that implement Comparator as you like
  - You can sort different data structures
    - Construct/sort TreeSet or TreeMap using the comparator you want
  - For example, sort Students by score *or* by name

Sorting differently

- Suppose you have students sorted by *score*, in a TreeSet you call studentsByScore
- Now you want to sort them again, this time by *name*
  - *Create the following Comparator*

```
public class StudentByNameComparator implements
Comparator<Student> {
          @Override
          public int compare(Student s1, Student s2) {
          return s1.name.compareToIgnoreCase(s2.name);
           }


}
```

Sorting differently

Add to the Main Method:

```
TreeSet<Student> setByName = new
TreeSet<Student>(new StudentByNameComparator());

setByName.addAll(set);

iter = setByName.iterator();

System.out.println("\nStudents by Name");

while (iter.hasNext()) {

      Student s = iter.next();

      System.out.println(s.name + "  " + s.score);

      }

}
```

Solution

- See this solution in the examples GitHub Repo…

https://github.com/fxwalsh/data-struct-algo-2017-examples.git