COP 3330, Spring 2013

Sorting with the Comparable interface

Instructor : Arup Ghosh
02-22-13

School of Electrical Engineering and Computer Science
University of Central Florida

The Comparable interface

- Defined already in `java.util`.
- Declares just one method – `compareTo()`
- If a class implements `Comparable`, its objects can be compared among themselves.
  - Less than, greater than or equal.

Why use it?

- The method `Collections.sort()` will sort a List.
- Note – List is an interface. AbstractList, ArrayList, LinkedList, Vector classes implement it.

  Visit - http://docs.oracle.com/javase/1.4.2/docs/api/java/util/List.html
- Note: The method `Arrays.sort()` will sort arrays, both arrays of primitive types and object types.

  (Visit - http://docs.oracle.com/javase/1.4.2/docs/api/java/util/Arrays.html)
- But, to sort you must be able to compare objects.
- Pretty easy for numbers, but what if you have an arbitrary class? Say a `Dog[]` array?
- How do you sort this? name? age?
- `Collections.sort()` cannot figure this out on its own.

Comparable and sorting

- To get around this, Collections.sort() will only accept List of types that implement the Comparable interface.
- Comparable types are guaranteed to have a compareTo() method, and this can be used for comparison.
- Note - a sorting method that doesn’t itself know how to sort its input.
- Because the *input* knows how to compare itself.

Strings

- The String class implements the Comparable interface.

- Make an ArrayList of String and call Collections.sort()
  - Immediately sorts in lexicographic order.
- Why lexicographic? Because that’s how the compareTo() method for String compares lexicographically.

compareTo()

- `Arrays.sort() or Collections.sort()` expects `compareTo()` to behave in a certain way.
- When comparing objects a and b, we call `a.compareTo(b);`
- This should return an int that is:
  - Negative, if a < b in our preferred ordering
  - Positive, if a > b in our preferred ordering
  - 0, if a = b in our preferred ordering.
- Not symmetric!
  `a.compareTo(b)` will be the *negative* of `b.compareTo(a)`.

Comparable is parameterized

- In the documentation, you'll see that String implements the interface *Comparable<String>*

- Triangular brackets contain a *type parameter*.
  - We'll cover this in detail when we do generics.
- In general, you will always match the types:

```
class Dog  implements Comparable<Dog> {
   …
   // The comparison function demanded by Comparable
   public int compareTo(Dog other) {
     …
   }
}
```

The subtraction metaphor

- You can imagine that `a.compareTo(b)` just returns `a – b`, as if the two were numbers.

- Thinking this way has two advantages:
  - You can always remember which way the signs go.
  - You can actually use subtraction to simplify your `compareTo()` code.

- Let’s do that in the Dog class example. We will add name, age, getDogName(), getDogAge() and constructors to that class.

Under the hood?

- What's really happening in `Collections.sort()`?
- When it compares `i`th `element of arrayDog` and `j`th `element of arrayDog`, it does this:

```
Comparable<Dog> a = new Dog(“Tommy”, 12);// Polymorphic
Comparable<Dog> b = new Dog(“Mega”, 5);    // Assignment
int c = a.compareTo(b);
…
// Remaining sorting logic (swapping etc)
```

Summary

- `Comparable<T>` is the interface the class `T` implements to indicate that its objects can be compared among themselves.

- **java.util.Collections.sort(List)** and **java.util.Arrays.sort(Object[])** me thods can be used to sort using natural ordering of objects.

- This is an order of magnitude faster than simple sorting algorithms like bubble sort or insertion sort.

- Also saves the trouble of writing a sort routine for every class.