Collections min method

min(..) method is overloaded method in Collections class. Previously we had studied methods like [swap(..)](http://data-structure-learning.blogspot.com/2015/06/collections-class-swap-method.html), [synchronized](http://data-structure-learning.blogspot.com/2015/06/synchronized-collections-by-collections.html), [binarySearch(..)](http://data-structure-learning.blogspot.com/2015/06/collections-class-binarysearch-method.html), [copy(..)](http://data-structure-learning.blogspot.com/2015/06/collections-class-copy-method.html), [fill(..)](http://data-structure-learning.blogspot.com/2015/06/collections-class-fill-method.html) and [addAll(..)](http://data-structure-learning.blogspot.com/2015/05/collections-class-addall-method.html).

To understand this method you need to know the [Comparable](http://data-structure-learning.blogspot.com/2015/06/comparable-interface-for-sorting.html) and [Comparator](http://data-structure-learning.blogspot.com/2015/06/comparator-interface.html) Interfaces and [differences between them](http://data-structure-learning.blogspot.com/2015/06/difference-between-comparable-and.html). I would highly recommend that you read those posts.

There are two versions of min(..) method. In this post we will cover only one of them. In next post we will discuss other one. The difference is that one method takes Comparable<T> interface and another takes Comparator<T> Interface as natural ordering method.

First we will cover the method that expects us to write Comparable<T> interface. All elements must have implemented Comparable<T> interface and they must be comparable i.e. e1.compareTo(e2) must not throw ClassCastException for any elements in collection. We will take 2 examples for this method. One in which class (Integer, String) has Comparable implementation provided and another in which we will implement our own Comparable interface.

Below is the code for min(..) method in Collections class.

**public** **static** <T **extends** Object & Comparable<? **super** T>> T min(

Collection<? **extends** T> coll) {

/\*\*

\* Take iterator of the collection

\* \*/

Iterator<? **extends** T> i = coll.iterator();

//This statement is used for comparison with he candidate.

T candidate = i.next();

**while** (i.hasNext()) {

T next = i.next();

/\*\*

\* next and candidate are compared.

\* if next is small it becomes candidate for comparision

\* for next elements.

\* \*/

**if** (next.compareTo(candidate) < 0)

candidate = next;

}

**return** candidate;

}

If you see the method signature <T **extends** Object & Comparable<? **super** T>> it seems to be confusing. But it is really not. Let us extract bits from it and understand it.

Comparable<? **super** T> - ? super T is wild card used so as to compare super type of T.

T **extends** Object - this means that type T must extend the class Object. This is used because of backward compatibility.

T **extends** Object & Comparable<? **super** T> - combining it together type t must extend class Object and Comparable of super type of T.

For class String and Integer, they have Comparable<T> implemented in it. We do not need to do much for finding min for it.

**public** **static** **void** minDemo() {

List<String> list = *populateLanguages*();

System.***out***.println(list);

String str = Collections.*min*(list);

System.***out***.println("min is "+str);

List<Integer> ints=**new** ArrayList<Integer>(Arrays.*asList*(6, 5, 4, 3, 3, 2, 5, 8));

**int** min=Collections.*min*(ints);

System.***out***.println(ints);

System.***out***.println("min is "+min);

}

Output:

[Java, JavaScript, C#, Python]

min is C#

[6, 5, 4, 3, 3, 2, 5, 8]

min is 2

Now we will find min using our own Comparable<T> interface and Person class.

Consider the Person class below.

**package** org.collections;

**public** **class** Person **implements** Comparable<Person>{

**private** **int** age;

**private** String name;

**public** Person(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (obj **instanceof** Person) {

Person o = (Person) obj;

**if** (o.getAge() == **this**.getAge() && o.getName().equals(**this**.getName())) {

**return** **true**;

}

**return** **false**;

}

**return** **false**;

}

@Override

**public** **int** hashCode() {

StringBuilder sb = **new** StringBuilder();

sb.append(**this**.getAge()).append(**this**.getName());

**return** sb.hashCode();

}

/\*\*

\* compareTo(T o) is used as natural comparison method where

\* comparison is done on lexical order on Person's name.

\* \*/

@Override

**public** **int** compareTo(Person person) {

**return** **this**.getName().compareTo(person.getName());

}

@Override

**public** String toString() {

**return** **new** StringBuilder()

.append("{")

.append(**this**.getName())

.append(" ")

.append(**this**.getAge())

.append("}")

.toString();

}

}

Now we write method to find min for Person class. Remember we wrote our Comparable<T> interface in Person class for String type. So the minimum for String is based on Lexical Order.

**public** **static** **void** minComparableDemo(){

List<Person> list=**new** ArrayList<Person>();

list.add(**new** Person("Eddard",55));

list.add(**new** Person("Rob",23));

list.add(**new** Person("Joffery",21));

list.add(**new** Person("Sansa",19));

list.add(**new** Person("Rickon",7));

list.add(**new** Person("Brandon",9));

System.***out***.println("Person list is "+list);

Person p=Collections.*min*(list);

System.***out***.println("min is "+p);

}

Output

Person list is [{Eddard 55}, {Rob 23}, {Joffery 21}, {Sansa 19}, {Rickon 7}, {Brandon 9}]

min is {Brandon 9}