24 Is the Data Sorted?

Introduction

Once we have designed several sorting algorithms it would help if we had a method that determined whether the given collection of data (a list of *Objects* or an *ArrayList* is sorted.

We know by now that we can design such methods to work for a number of possible kinds of data collections and a number of different ways we may choose for comparing the elements of the data collection.

Let us start with some examples, sorting books by titles, year of publication and the author's name.

```
Book sn  = new Book("AP", "SN", 1996);
Book akm = new Book("RPW", "AtKM", 1956);
Book eoe = new Book("JS", "EoE", 1954);
Book atf = new Book("AM", "AtF", 1962);
Book mls = new Book("PC", "MLS", 2002);

ALoObj authors = new ConsLoObj(atf,
                        new ConsLoObj(sn,
                        new ConsLoObj(eoe,
                        new ConsLoObj(mls,
                        new ConsLoObj(akm, new MTLoObj())))));

ALoObj titles =  new ConsLoObj(akm
                        new ConsLoObj(atf,
                        new ConsLoObj(eoe,
                        new ConsLoObj(mls,
                        new ConsLoObj(sn, new MTLoObj())))));

ArrayList years = new ArrayList();

public void initYears(){
  this.years.add(this.eoe);
  this.years.add(this.akm);
  this.years.add(this.atf);
  this.years.add(this.sn);
  this.years.add(this.mls);
}
```

We have two lists of *Objects* and one *ArrayList*, presumably results of different sorting algorithms. But in order to determine whether the data is sorted, all we need is to be able to examine the data elements one at a time. That means, all we need is an iterator that traverses the data and generates the elements in the order in which they appear in the original data structure. That means that an *IRange* will be one of the method arguments.

For our examples, we define three objects of the type IRange:

```
IRange authorsIt = new ListRange(authors);
IRange titlesIt  = new ListRange(titles);
IRange yearsIt   = new ArrayListRange(years, 0);
```

The three sets of data shown here are sorted according to three different criteria: by title, by author's name, and by the publication year. That means, our sorting checker has to know what method was used to sort the data. We use the *Comparator* interface to define how the comparison is made, i.e. using the method

```
interface Comparator{
  public boolean compare(Object obj1, Object obj2);
}
```

We can now formulate the purpose statement and the header:

```
// determine whether the data generated by the given iterator
// is sorted according to the ordering given by the comparator
boolean isSorted(IRange it, Comparator comp){
...
}
```

Our next step is to define three classes that implement the *Comparator* interface to perform the three different kinds of comparisons of books: by title, by author, and by year of publication:

```
public class ByTitle implements Comparator{
  // compare two books by their title
  public boolean compare(Object obj1, Object obj2){
    return ((Book)obj1).title.compareTo(((Book)obj2).title);
  }
}
```

```
public class ByAuthor implements Comparator{
  // compare two books by the name of the author
  public boolean compare(Object obj1, Object obj2){
    return ((Book)obj1).author.compareTo(((Book)obj2).author);
  }
}

public class ByYear implements Comparator{
  // compare two books by their year of publication
  public boolean compare(Object obj1, Object obj2){
    return ((Book)obj1).year − ((Book)obj2).year;
  }
}
```

Of course, we also need three obejcts of the type *Comparator*:

```
Comparator byTitle  = new ByTitle();
Comparator byAuthor = new ByAuthor();
Comaprator byYear   = new ByYear();
```

The three classes that implement the *Comparator* will be defined in separate files, while all the rest of the code shown here will be a part of the *Examples* class. We can write down trests for these classes:

```
byTitle.compare(akm, sn) −−−> true
byTitle.compare(mls, eoe) −−−> false
byAuthor.compare(sn, akm) −−−> true
byAuthor.compare(eoe, mls) −−−> false
byYear.compare(sn, mls) −−−> true
byYear.compare(sn, atf) −−−> false
```

We can now make examples of the invocation of the method *isSorted* and show the expected results:

```
test("sorted by title",  true,  isSorted(titlesIt,  byTitle);)
test("sorted by author", true,  isSorted(authorsIt, byAuthor));
test("sorted by year",   true,  isSorted(yearsIt,   byYear));
test("sorted by title",  false, isSorted(authorsIt, byTitle));
test("sorted by year",   false, isSorted(titlesIt,  byYear));
test("sorted by author", false, isSorted(yearsIt,   byAuthor));
```

We can now look at the template for this method. The method will be defined within our *Examples* class, or within the *Algorithms* class, but the main point is that it makes no use of the instance of the class that invokes it. Therefore, there are no *...this.−−−...* fields in the template. The only pieces of data needed for the computation are the two arguments. That means our template will have the following elements:

```
... it.hasMore() ...
    (if above produces true)
    ... it.current() ...
    ... it.next() ...
    ... this.isSorted(it.next(), comp)


... comp.compare(−−−, −−−)
```

If *it.hasMore*() returns false, there is no data in the data set and so it is sorted by default. We also realize that the data is also sorted if there is only one element in the data set, but for now we put that aside. However, looking at the template further, we see a problem. The method *comp.compare* requires two arguments, but we only have one *Object* available. Thinking about the cause of the problem we see that just looking at the first element of the data set is not sufficient — we need to compare the first with the first in the remainder of the data set. We decide to define a helper method that receives the element to compare to as an additional argument (an accumulator). The method header and the purpose become:

```
// is the data set given by it sorted and are all its elements
// before obj with regard to comp
boolean isSortedAcc(IRange it, Comparator comp, Object acc){...}
```

Let us illustrate this visually:

```
isSorted(it, comp)      isSorted(it, comp, acc)
+---+---+---+---+        +---+ +---+---+---+
| 3 | 5 | 2 | 7 |        | 3 | | 5 | 2 | 7 |
+---+---+---+---+        +---+ +---+---+---+
    IRange it             acc   IRange it
```

The example illustrates three points. First, it is clear how to complete the body of the original method:

```
// determine whether the data generated by the given iterator
// is sorted according to the ordering given by the comparator
boolean isSorted(IRange it, Comparator comp){
   if (it.hasMore())
      return isSortedAcc(it.next, comp, it.current());
   else
      return true;
}
```

Next, we see that in our method we must make sure that *acc* is less that or equal to the first element of the structure traversed by *it*, and that we still must make sure that the rest of the list is sorted. Of course, *acc* is added to the template and becomes the second argument to the *comp.compare*(...) method.

Here are the examples to illustrate the different possibilities:

```
ALoObj books1 = new ConsLoObj(akm,
                       new ConsLoObj(atf, new MTLoObj()));
ALoObj books2 = new ConsLoObj(akm,
                       new ConsLoObj(eoe, new MTLoObj()));

IRange brange1 = ListRange(books1);
IRange brange2 = ListRange(books2);

isSortedAcc(brange1, byYear, sn) ---> false
isSortedAcc(brange2, byAuthor, sn) ---> false
isSortedAcc(brange1, byYear, eoe) ---> true
```

We are ready for the template. It is just as for the *isSorted* method, but also includes *acc* that can be used as one of the arguments for the *comp.compare* methods.

The body becomes:

```
// is the data set given by it sorted and are all its elements
// before obj with regard to comp
boolean isSortedAcc(IRange it, Comparator comp, Object acc){
   if (it.hasMore()){
      return ((comp.compare(acc, it.current()) <= 0)
            && isSortedAcc(it.next, comp, it.current());
   }
   else
      return true;
}
```

We can now run all of our test cases. The class diagram for these classes is shown below - we use a dotted line to indicate that a method consumes an instance of another class or interface type.

```
// Designing isSorted method:

        +----------------------------------------------+
        | boolean isSorted(IRange it, Comparator comp) |...........
        +----------------------------------------------+     :    :
               ...............................................    :
               :                                                  :
               v                                                  v
     +-------------------+                     +-----------------------------------+
 +- >| IRange            |< - - - -+           | Comparator                        |
 |   +-------------------+         |           +-----------------------------------+
     | boolean hasMore() |                     | int compare(Object o1, Object o2) |
 |   | Object current()  |         |           +-----------------------------------+
     | IRange next()     |                           ^            ^          ^
 |   +-------------------+         |                 |            |          |
 +- - - - -
          |                        |                 |            |          |
  +-----------------+      +---------------+    +---------+ +----------+ +--------+
  | ArrayListRange  |      | ListRange     |    | ByTitle | | ByAuthor | | ByYear |
  +-----------------+      +---------------+    +---------  +----------+ +--------+
  | ArrayList alist |-+  +-| ALoBook alist |        :             :          :
  | int current     | |  | +---------------+        :             :          :
  +-----------------+ |  |                          :             :          :
           +----------+  +-------+  +-----------+   :             :          :
           |                     |  |           |   :             :          :
           v                     v  v           |   :             :          :
  +-----------------+        +---------+        |   :             :          :
  | ArrayList       |        | ALoBook |        |   :             :          :
  +-----------------+        +---------+        |   :             :          :
+-| (... Book ...)  |            / \            |   :             :          :
| +-----------------+            ---            |   :             :          :
|                                 |             |   :             :          :
|                       ----------------        |   :             :          :
|                       |              |        |   :             :          :
|                +----------+  +--------------+ |   :             :          :
|                | MTLoBook |  | ConsLoBook   | |   :             :          :
|                +----------+  +--------------+ |   :             :          :
|                   +----------| Book first   | |   :             :          :
+-------------+     |          | ALoBook rest |-+   :             :          :
              |     |          +--------------+     :             :          :
              |     |   ......................................................
              v     v   v
          +---------------+
          | Book          |
          +---------------+
          | String title  |
          | String author |
          | int year      |
          +---------------+
```