# 8 Loops; Sorting

#### Goals

In the first part of the lab you will learn how to convert recursive loops to imperative (mutating) loops using either the **Java** while statement or the **Java** for statement to implement the imperative loops.

In the second part we will look at how we can leverage the direct access to the items within the data set to implement a new kind of sorting algorithm.

For this lab download the files in *Lab9-Sp2010.zip*. The folder contains the following files:

- The file Balloon.java our sample data class
- The file ISelect. java the interface for a generic predicate method
- The files RedBallon and SmallBalloon that implement the ISelect interface for the Balloon data.
- The files IList.java, MTList.java, and ConsList.java that define a generic cons-list that implements the *Traversal* interface.
- The file ArrListTraversal. java shows how we can define a *Traver*-**l** sal wrapper for the ArrayList class.
- The Algorithms. java file shows an implementation of several algorithms that consume data generated by a Traversal iterator and illustrates a number of ways in which loops can be implemented in Java.
- The Examples. java file that defines examples of all data and defines all tests.

Create a new **Project** *Lab9* and import into it all files from the zip file. Import the tester.jar and colors.jar.

#### 8.1 Converting Recursive Loops into Imperative while Loops

The goal of this part of the lab is to make sure you know how to implement a traversal over data within an ArrayList using Java while and for

loops. Make sure you understand the role of each part of the loop method definition: *BASE VALUE*, *CONTINUATION-PREDICATE*, *CURRENT*, *ADVANCE*, *UPDATE*, and know how to construct both the while loop and the for loop.

- Work with the Lab handout. The first page gives you an overview
  of all classes and interfaces and the relationship between them. We
  introduce a dotted line from a method that consumes an instance of
  some class to that class.
- Read first the code for the contains method and for the countSuch method in the Algorithms class. These have been designed in the classical HtDP style.
- We will look together at the next two examples of orMap in the Algorithms class.

We first write down the template for the case we already know — the one where the loop uses the Traversal iterator. As we have done in class, we start by converting the recursive method into a form that uses the accumulator to keep track of the knowledge we already have, and passes that information to the next recursive invocation.

Read carefully the *Template Analysis* and make sure you understand the meaning of all parts.

```
TEMPLATE - ANALYSIS:
return-type method-name(Traversal tr){
// invoke the methodAcc: | acc <-- BASE-VALUE |
                        +----+
  method-name-acc(Traversal tr, BASE-VALUE);
return-type method-name-acc(Traversal tr, return-type acc)
 ... tr.isEmpty() ...
if true:
 ... acc
                                           -- return-type :: BASE-VALUE
if false:
 ... | tr.getFirst() | ...
                                           -- E
                                                          :: CURRENT
 ... update(T, return-type)
                                           -- return-type :: UPDATE
 i.e.: ... | update(tr.getFirst(), acc) | ...
 ... | tr.getRest() |
                                           -- Traversal<T> :: ADVANCE
 ... method-name(tr.getRest(), return-type) -- return-type
 i.e.: ... method-name-acc(tr.getRest(), update(tr.getFirst(), acc))
```

Based on this analysis, we can now design a template for the entire problem — with the solution divided into three methods as follows:

## Understanding orMap

• Look at the first two variants of the orMap method (the recursively defined variant and the variant that uses the while loop. Identify the four parts (BASE-VALUE, Termination/Continuation PREDICATE, UPDATE, and ADVANCE) in each of them.

Look also at the tests in the Examples class.

- After you understand how the while loop works, design two variants of the method that produces a new ArrayList that contains all elements of the original list that satisfy the given ISelect predicate.
   Test the methods by producing all red balloons or all small balloons.
- Design and test two variants of the andMap method that determines whether all elements of a given list satisfy the given ISelect predicate.

Test the methods by checking whether a list contains all red balloons or all small balloons.

## Converting while loops into for loops

Repeat all the parts of the previous task with the remaining two variants of the orMap — namely the one that uses the for loop with the Traversal and the one that uses *counted* for loop.

#### For Each

Optionally, you may look at the ultimate abstraction of these traversals shown in the ForEach class.

 Read the tests for for each variant of the compute method of the ForEach class shown in the Examples class. Make sure you understand how they work. Design additional tests for each of the three compute methods.

## 8.2 Sorting

**Selection sort** is one of the familiar sorting algorithms. It is well suited for the situations where you are trying to minimize the moving of the data from one location to another.

Suppose you have an ArrayList of data of size s in which the first k elements are sorted, and every item in the unsorted part is greater than the largest item in the sorted part.

You would like to sort the rest of the ArrayList. We know how to swap two items in the ArrayList. So, if we can find the location of the smallest item in the unsorted part and swap it with the first item in the unsorted part, the sorted part will be one item bigger, and the unsorted part will be one item smaller.

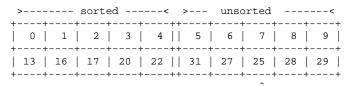
If we repeat this until the last item is swapped into its correct position, we will have finished sorting the remainder of the ArrayList.

Here is an example:

> sorted< >																		
	0	1	.	2		3		4		5		6		7		8	!	9
	13	16	5	17		20		27		31		22	2	25	2	8	2	9
+-			-+		- + -		-++		- +		+-	^	+					+

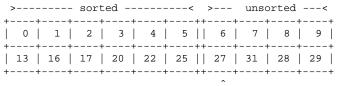
min unsorted

Swap elements at locations 4 and 6:



min unsorted

Swap elements at locations 5 and 7:



min unsorted

Swap elements at locations 6 and 6:

> sorted >- unsorted													
	0	1	2	3	4	5	6	7	8	9			
	13	16	17	20	22	25	27	++    31 ++	28	29			
7			,		,		,		n unsc				

What about the case when none of the ArrayList is sorted? Well, then the sorted part has size 0, and the unsorted part starts at the index 0.

- 1. In the Algorithms class design the helper method findMinLoc that finds the location of the smallest item in the unsorted part of the given ArrayList.
  - *Note:* Think carefully through the first step of the *design recipe*, to make sure you know what the method consumes and what does it produce.
- 2. In the Algorithms class design the method selectionSort that implements the *selection sort* algorithm.
- 3. Design two Comparators for the Balloons, the BalloonsBySize that compares the balloons by their radius, and BalloonsByHeight that compares them by their distance to the top.
- 4. Test your sorting method and the helper method on lists of balloons using each of the two Comparators.