# 9 Abstracting Traversals; Using Runtime Exceptions; Introducing Collections

In this problem set you will work with several predefined classes that represent cities and lists of cities, as well as files that provide infrastructure for the tests and for dealing with user input.

The focus of your work is on learning to use abstractions in building reusable program components.

**Part 1: Iterators, Loops, User input**

Start by downloading the file **HW9part1.zip** and making an Eclipse project **HW9part1** that contains these files. Add **jpt.jar** as a *Variable* to your project. Run the project, to make sure you have all pieces in place. The main method is in the class *Interactions*.

The classes you will work with are the following:

- **class** *City* represents name, state and a zip code for one city
- *AListOfCities* and its subclasses represent lists of cities
- **interface** *IRange* to represent an iterator
- **class** *Examples* is the class that holds examples of data and the tests for all your methods
- **class** *Interactions* is the class that facilitates user interactions and allows you to explore the behavior of parts of your program
- **class** *Algorithms* contains methods that implement loops, such as our *orMap* and filter as well as other algorithms, such as sorting algorithms.
- **interface** *ISelect* and **interface** *IObj2Obj* represent function objects consumed by the loop methods.
- **interface** *ISame* is our standard interface for implementing the usual extensional equality comparison of objects

### 9.1 Problem

For the given classes *City*, *AListOfCities* (and its subclasses), and the interface *IRange* design the following classes and methods:

1. **class** *ListRange* that implements the *IRange* iterator for the list of cities.

2. In the **class** *Algorithms* design the method *buildList* that consumes an IRange iterator and produces a list of cities. In one of your tests for this method make a copy of a list of cities and compare the results. (You may need to reverse the result.)

3. Explore the use of the method *buildList* in the **class** *Interactions* by using the *InGuiRange*, *InConsoleRange* and *InFileRangeBuffered* iterators.

4. Design the method *orMapWhile* that is a variant of *orMap* that uses Java *while* statement instead of recursion. Use as a model the implementation of the filter and *filterWhile* methods.

5. Add the methods *andMap* and *andMapWhile* to the *Algorithms* class.

6. In the *Examples* class include the following additional tests for the filter, orMap, and andMap methods:

   - produce a list of all cities in a given state
   - find out whether there are any cities in a given state in some list of cities
   - are all cities in some list in a given state
   - is there a city with the given name in this list
   - produce a list of all cities with the given name from some list of cities
   - do all cities in some list have the given name ∎

### 9.2 Problem

Design the **class** *ArrayListRange* to provide a traversal of the data represented by Java *ArrayList* object.

Re-run the tests from the Part 5 of the previous problem using data stored in an ArrayList. ∎

### 9.3 Problem

Design a new variant of the **class** *ArrayListRange* that will allow the user to decide at construction time what part of the given *ArrayList* should the iterator traverse. So, for example, you may choose to only look at the consecutive elements of the *ArrayList* starting at index 5 and going up to index 14.

Re-run the tests from the Part 5 of the previous problem using data stored in an ArrayList. ∎

### 9.4 Problem

The goal here is to start thinking about systematic design of tests.

Read the code for the **class** *SimpleTestHarness*. Design a test suite for it.

Write a pragraph or two of documentation that describes what kinds of tests can/cannot be done using this test harness. ∎

## Part 2

In this part you will implement several sorting algorithms possibly using the iterators from the first part for the traversal. Use function objects that implement Java *Comparator* interface to determine the ordering of two objects.

### 9.5 Problem

Implement the insertion sort in the **class** *Algorithms* and test it on both a list of cities and an *ArrayList* that represents cities. ∎

### 9.6 Problem

Implement the quick sort in the **class** *Algorithms* and test it on both a list of cities and an *ArrayList* that represents cities. ∎

### 9.7 Problem

The selections sort works as follows. Select the smallest item in the list and remember its position. *Swap* the first item with the smallest one you found. The first item is now in the correct position. Repeat these steps for the remaining part of the list, until all items move into their correct places.

- Design the method that finds the index of the smallest item in the given array, traversed using the IRange iterator.
- With the given helper method, using the second variant of your iterator for ArrayList, design the selection sort for ArrayList. ∎

**Part 3**

### 9.8 Problem

We have the following data definitions:

A *Reply* consists of

- String *question*
- *ArrayList answers* that contains at least three elements, each of them also a String

A *CrystalBall* consists of an *ArrayList* of *Reply*

Make a new project named *Eliza*, import the files from *eliza.zip*, and your iterator classes for *ArrayList*

Add the *jpt.jar* variable to your project as well.

Now design the following methods

- in the class *Reply* design the method *randomAnswer* that produces at random one of the answers available
- in the class *CrystalBall* design the method *findReply* that consumes a String *s* and produces the *Reply* such that the *question* is a prefix of the given String, or a special *default Reply* object, if it finds no match.

  The method *startsWith* in the class String helps here.

Make an example of a *CrystalBall* object in the *Interactions* class.

In the *Interactions* class design a method that forever requests a String *s* from the user and displays the answer given by the *findReply* method fo the *CrystalBall* class. When the user aborts the input response, the method prints `Goodbye`. ∎