7 Abstracting over the Data Type; Java Documentation; Direct Access Data Structures

This lab consists of three parts.

In the first part you will learn how to abstract over the type of data a collection of data represents.

In the second part we will look at how Java documentation represents the classes and methods and will look at a couple of classes in the Java Collections Library.

In the third part we will work briefly with a data structure that allows us to access directly a specified item in the collection.

7.1 Abstracting over the Data Type

The goal of this section of the lab is to understand how we can design a more general programs by defining the common behavior for structured data, such as lists or binary trees, using parametrized data types.

Begin by downloading *lab8.zip* and building a project that contains all the files as well as the latest version of the *tester.jar*.

Your project should have the following files:

- *BookBST.java*
- *AcctBST.java*

A. Each file represents a complete program that deals with binary search trees. Set up two *Configuration*s to run each of them and run them.

B. In Eclipse, *Window menu −* > *New Window* will open a new window. Set up the new Eclipse window to show Java Perspective by selecting *Window menu −* > *Open Perspective −* > *Java*. Open the two files in the two windows in full size, side by side. Now observe the differences and similarities.

C. Copy the file **AcctBST.java** and add it to the project with the name **BST.java**. We now have two copies of class and interface definitions.

   Comment out the class definition for the class `Acct`. The one defined originally will be used again.

   Now replace `Acct` with `<T>` in all places that define the data type. So,

- `ABSTAcct` becomes `ABST<T>`
- `LeafAcct` becomes `Leaf<T>`
- `NodeAcct` becomes `Node<T>`
- `ICompAcct` becomes `IComp<T>`

Rename the `ExamplesAcctBST` class to `ExamplesBST`.

D. What else needs to be done? In the classes `ABST`, `Leaf`, and `Node`, in every place where we refer to `Acct` replace this with `T`.

E. We are almost done. Look at what still needs to be done. How will you deal with the similarities between the definitions of `ICompAcct` and `ICompBook`? Figure out this part.

F. The last part is to make the necessary changes in the `ExamplesBST` class. Here we need to specify what type of data will the binary search tree contain. So, the type `ABSTAcct` becomes `ABST<Acct>` indicating that we are dealing with the abstract class `ABST` with the type argument `Acct`. Finish the changes until there are no errors or warnings. Run the tests.

G. Copy the data definitions and tests from the `ExamplesBookBST` class, make the necessary changes, and run these tests as well.

7.2 Java Documentation

So far our purpose statements were sufficient for someone trying to understand how our program works and where to make changes, if another person wants to improve the program we have written. However, if we design a program that represents a reusable parametrized data type, such as our lists or binary search trees parametrized over the type of data they represent, the user of the code may not be interested in all the details of the implementation, but only the fields that she may be able to access, the constructors that can be used, and methods that can be invoked or overridden.

In general, most of the modern general purpose languages come with a special language for writing the purpose statements. The statements are then translated into cross-referenced web pages that allow the programmer to inspect the library without looking at the actual code.

JavaDocs basics

A. Go to the *javalib* web site at **http://www.ccs.neu.edu/javalib**. Go to the *Tester* tab, then look at *JavaDocs* tab and open the documentation for the latest version of the *tester* library. The web site you see has the documentation for all public fields and methods in the entire library. Click on the `Tester` tab on the left and you will see a description of the class `Tester`.

B. Scroll through the descriptions of the methods until you find `checkInexact`. Click on the method — and you will see the detailed description of the method - its purpose, its parameters, and the return value it produces.

C. Now look at the method `checkRange` in the *Method Summary* section. You can see that there is a number of methods with this name, some that consume an argument of the type `java.lang.Comparable<T>`, some that consume an argument of the type `java.util.Comparator<T>`.

D. These are two interfaces defined in Java libraries. The first is a part of Java main language package (`java.lang`. The classes and interfaces defined there are automatically imported to every Java program. For example, the class `String` is specified in the documentation as `java.lang.String`. We have used it all along without the need for any `import` statements.

E. However, the interface `java.util.Comparator<T>` is a part of the **Java Collection Framework** package (`java.util`), a library of classes and interfaces for dealing with collections of data.

Java Collections Framework

Go to the web site for Java libraries at: **http://java.sun.com/javase/6/docs/api/**.

A. Scroll through the *All Classes* frame on the left till you find `Comparable` and `Comparator`. You can see in the description that there is a lot of detail in there, much more than we would expect from such a simple function object. We will address some of these issues in the lectures.

B. It looks like we could replace our `interface IComp<T>` for the binary search trees with the `interface Comparator<T>`. Do it. You will need to add the `import java.util.*;` statement at the beginning of your program. Otherwise, the program should work as before.

ArrayList

Scroll through the *All Classes* frame on the left again, till you find `ArrayList`. The lecture handout included some of the methods one can use when manipulating a data collection that is represented by an `ArrayList`. Use the handout or the online documentation as you work on the last part of the lab.

7.3 Direct Access Data Structures with Mutation

For this part of the lab download the following files:

- The file *Balloon.java* — our sample data class
- The file *TopThree.java* will be used to practice working with `ArrayList` in imperative style (using mutation).
- 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.** and *colors.** libraries.

In this part of the lab we will work on lists of balloons, using the Java library class `ArrayList`.

Here are some of the methods defined in the class `ArrayList`:

```
// how many items are in the collection
int size();

// add the given object of the type E at the end of this collection
// false if no space is available
boolean add(E obj);

// return the object of the type E at the given index
E get(int index);

// replace the object of the type E at the given index
// with the given element
// produce the element that was at the given index before this change
E set(int index, E obj);
```

Other methods of this class are `isEmpty` (checks whether we have added any elements to the `ArrayList`), `contains` (checks if a given element exists in the `ArrayList` — using the `equals` method).

7.4 Using the ArrayList class

Notice that, in order to use an `ArrayList`, we have to add

```
import java.util.ArrayList;
```

at the beginning of our class file.

The first method you design will be within the class `TopThree`. The remaining methods will be defined within the `Examples` class. Of course, the tests for all methods will still be inside the `Examples` class.

A. The class `TopThree` now stores the values of the three elements in an `ArrayList`. Complete the definition of the `reorder` method. Use the previous two parts as a model. Look up the documentation for the Java class `ArrayList` to understand what methods you can use.

   Do not forget to run your tests.

B. Design the method `isSmallerThanAtIndex` that determines whether the radius of the balloon at the given position (index) in the given `ArrayList` of `Balloon`s is smaller than the given limit.

C. Design the method `isSameAsAtIndex` that determines whether the balloon at the given position in the given `ArrayList` of `Balloon`s has the same size and location as the given `Balloon`.

D. Design the method `inflateAtIndex` that increases the radius of a `Balloon` at the given index by 5.

E. Design the method `swapAtIndices` that swaps the elements of the given `ArrayList` at the two given positions (indices).

Note 1: We have used the words *position* in the `ArrayList` and *index* in the `ArrayList` interchangeably in the previous descriptions of tasks. Both are commonly used and we wanted to make sure you get used to both ways of describing an element in an `ArrayList`.

Note 2: Of course, the tests for these methods will also appear in the `Examples` class. Make sure that every test can be run independently of all other tests. To do this, you must initialize the needed data inside of the test method, evaluate the test by invoking the appropriate `checkExpect` method, and reset the data to the original state after the test is completed.

Note: Finish this lab and include your work in your portfolio.