**Lab 3 – Using and implementing sets**

Please continue to work with your pair partner(s) on this lab.

Objectives:

* to use existing ADTs in the Java Collections Framework
* to construct and implement a complete ADT
* to use existing software components to support building new software
* to gain more insight into using arrays to build data structures

***Problem 1: Creating a worker application using sets***

Write a program that manages a set of workers (*Set<Worker>*) by completing the provided *WorkerManager* program.

Recall that your *worker* package sits inside your *JavaPackages* folder in your home directory so it will be accessible from this lab folder provided you use an import worker.\* statement; thus, keep the *WorkerManager.java* in your lab folder OUTSIDE *JavaPackages*. We’ll be saving *Worker* objects to an object file so make sure that interface *Worker.java* in package *worker* extends *Serializable* and contains an import java.io.\*; statement.  
  
In this lab, you will use one of Java's set ADT implementations called *HashSet<E>*. The complete documentation is available here <http://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html> but we mostly care about the *Set<E>* interface implemented by class *HashSet<E>*: <http://docs.oracle.com/javase/7/docs/api/index.html?java/util/Set.html>. Study this interface carefully as it contains functionalities available to you when using the *workers* instance variable in class *WorkerManager*. 

The *WorkerManager* program provides a complete template for your application, but many of the pieces remain to be filled in. These places are all marked with comments in the program. The add method is complete, but it calls three auxiliary/helper methods: newVolunteer, newHourlyEmployee and newSalariedEmployee that are not complete. Complete these three methods first as described.

Next, partially complete the report method by having it generate a listing containing detailed information on ALL the workers currently in the workers set. (*For now*, you do not need to prompt the user for anything, since the method generates a report containing all workers.) We’ll modify this method later on.  
   
At this point you should be able to compile and run the program as it stands so far. You should be able to add any type of worker and list all the workers in the set. Make sure you have this much working before you go on.   
   
Once the add and preliminary report methods are working, go on to do the remove method. In order to remove a *Worker* from the *workers* set, you can use an enhanced for loop to find the correct *Worker* (by name) and then pass the matching object to the set’s remove method. If your program produces a ConcurrentModificationException, you may want to break from the enhanced *for* loop (via a break statement) once you remove the match from the set.

Note that this implementation of the remove method is not the most efficient. A better approach would be to use a temporary *Worker* object ***equal*** to the one you want to remove (i.e. has the same name):

Worker dummy = new Volunteer(name);

Simply pass *dummy* to the set’s remove method, and it will remove a matching worker if it contains one. However, such an implementation requires overriding the hashCode() method from class Object which is beyond the scope of this lab; we will come back to this topic later on.

Next implement the save and load methods. Note that the *HashSet<E>* class already implements interface *Serializable* which allows you to save it to an object file (as you did with a list in the last lab)

Finish the *WorkerManager* by completing the manage and report methods as well as any other parts you didn't get done so far. For the report method, use the add method as a template to prompt the user for the type of report desired. For any report other than all workers, you will need to iterate through the whole worker set to find the desired workers and report only those that match the search criteria (use the instanceof operator to determine the type of each worker). Allow the user to specify ANY type from the worker hierarchy including interfaces and abstract classes.

Class *WorkerManager* only manages properties common to all types of workers; specifics related to *Volunteers*, *HourlyEmployees* and *SalariedEmployees* need to be managed separately by their own manager classes. As a result, in order to complete the managemethod in class *WorkerManager*, you need to create three new Java classes: *VolunteerManager*, *HourlyEmployeeManager*, and *SalariedEmployeeManager*. Each of these should be designed to manage a single worker of the specified type with a user interface similar to that of the *WorkerManager*. Thus, each of the new manager classes will contain an instance field for the *Worker* object it will manage (i.e. *VolunteerManager* should have an instance field of type *Volunteer*)and will have a constructor with a parameter for its *Worker* object field so that all updates will be on the original object. In designing the three new classes, consider what types of management are appropriate for each type of worker based on the classes themselves. There will be some overlap, but they won't generally be the same. The manage method in class *WorkerManager* simply creates the appropriate type of manager class and calls it runManager method.

***Problem 2: Implementing your own ArraySet ADT***

This problem is the first of several in which you will construct your own class implementing an interface either provided to you in the lab folder or stored in the *zhstructures* package in */usr/people/classes/Java* (documentation accessible here: <http://www.users.csbsju.edu/~csweb/ZHStructures/index.html>)

To distinguish your own classes and interfaces from the ones in package *zhstructures*, you need to give them their own names and put them in your own package. You’ll use your initials to name the package and its associated classes and interfaces. For example, I (***I****mad* ***M******R****ahal*) would name my package ***imr****structures* and the class we're constructing today ***IMR****ArraySet*. Please note that names are case sensitive, package names are (by convention) all lower case, and interface and class names must start with an upper case letter.

For the rest of this labs(and several labs to come), when names in the write-up or in files begin with ***<FOO>*** or ***FOO***, replace the string with your upper case initials, and when they begin with ***<foo>*** or ***foo***, replace the string with your lower case initials. Always be consistent about what your initials are. When you’re working in a pair, implement the classes in the directories and with the initials of the person doing the typing. The other person can then get copies of these files for their own directories and change the initials in the files and the file names themselves to have the appropriate new initials. Use *find and replace*, and be sure to check Case-Sensitive Matching when you make these changes.

Create a package directory in your *JavaPackages* directory with the name of your package (***imr****structures* in my case). As you work, be sure to include appropriate javadoc documentation throughout this class file.

(If you’re using DrJava, you must have the corresponding class path directories entered; they're probably there already from Lab 1, but if you have problems with DrJava not finding *zhstructures* or things in your *JavaPackages* directory, use the procedures in the Lab 1 pre-lab to set things up. The TA or lab instructor will help you, if necessary.)

### *Part 2.1: Creating class <FOO>ArraySet to implement interface <FOO>Set*

### Move file *FOOSet.java* into your structures folder but keep the *FOOArraySeyTest.java* in your lab folder. Rename *FOOSet.java* with the (capitalized) initials you are using for your structures (e.g., I would name it *IMRSet)*, then open the file. Change all instances of “*<FOO>*” to the (capitalized) initials you are using for your structure classes; change the package declaration to the name of your structures package (*imrstructures* in my case).

### Repeat the same thing for the *FOOSetTest.java* file; rename it with the (capitalized) initials you are using for your structures (e.g., I would name it *IMRSetTest),* change the import <foo>structures.\* statement to your structures package, and again change all instances of “<*FOO*>” to the (capitalized) initials you are using for your structures.

Now create a new Java class called <*FOO>ArraySet* but use the (capitalized) initials you are using for your structures instead of *<FOO>*. This class must reside in the same package alongside file <*FOO>Set.java* (include a package … statement at the top of your class) and implement interface *<FOO>Set* (so include an implements clause in the class header). Import the following additional packages in your class: zhstructures.\*, java.io.\* and java.util.\*.

Be sure to make the class generic by putting a generic parameter <ElementType>immediately after the class name AND the implemented interface.

Next copy all methods from interface *<FOO>Set* to your class. Include a constructor with no parameters and a second constructor with an int parameter called initLength.

Methods that return a boolean can all return *true* for now, the size method can return *0* and the other methods can return null. Constructors need do nothing for now since we have not added any instance variables yet. When you complete these steps, your class should compile without errors, although there may be warnings about unused parameters.

Now you are ready to start implementing the *Set* ADT.

### *Part 2.2: Completing class <FOO>ArraySet*

Let's first consider how to build a set using an array to store its elements. An array has a fixed length, but a set may grow and shrink in size. We'll use an idea similar to the *DoubleTank* class for this implementation. Think of an array as a tank that can hold objects: its length corresponds to the capacity of the tank; we'll use a *size* variable, just like the tank used a *level* variable, to keep track of how many elements are in the set. We'll keep the elements of the set in slots *0* … *size−1* of the array.

To implement this scheme, we'll need two private instance variables—an ElementType[]array elements and an intvariable sizeinitialized to *0*—along with a public static final int class variable DEFAULT\_INIT\_LENGTH, initialized to a reasonable power of 2 such as 1024.

We'll turn now to implementing the methods needed for the array set. Every class needs at least one constructor; for the array set, we'll make two constructors that both create an empty set. The first will create a set with a capacity (array length) of DEFAULT\_INIT\_LENGTH; the second will create a set with a capacity (initLength) specified by the user. The first can be implemented by calling the second, passing the default length as the parameter (i.e. this(DEFAULT\_INIT\_LENGTH)); the second needs to allocate an Object array, cast it to ElementType[] and assign it to the elements instance variable. The size instance variable is already initialized to *0* at its declaration. Users can employ the second constructor when they know the maximum set size in advance or if they want to use a particular initial length; otherwise, they use the first constructor. Since the second constructor has a parameter, you should consider whether there are possible invalid values for the parameter, and if so, the constructor should throw an IllegalArgumentException if the parameter value is not valid. In this case, be sure to document the exception with an *@throws* tag in your Javadoc comments for the constructor (which, of course, you’ve already written).

Next, implement the isEmpty, size and contains methods. These should be straightforward. The contains method must search through the elements of the set until it either finds a match using the equals method or exhausts the elements of the set without finding a match.

For the add method, we can put the new element any place we want in the elements array that is part of the contiguous block of elements, but it's easiest to put it at the end of the block. Before adding the new element, however, we need to check whether it's already there (using the contains method). If the element is already in the set, the method should change nothing and return *false*. Once the method has determined that the element is not already present, it needs to check whether the array is at capacity (the *size* of the set is equal to the length of the elements array); if so, it calls the private reallocate method described in the next paragraph. Finally, the method stores the new element at index size in the array, increments size and returns *true*.

The reallocate method should be private, since it's a helper method only used internally by the class itself. It should create a new array of Object cast to ElementType[], twice the length of the elements array, and copy all objects from elements into the new array. Once it's done copying elements, it stores the reference to the new array in the elements instance variable and returns.

The remove method is a little trickier than add. We need to check if the element to be deleted is actually there. We could use the contains method to do that, but if it returns *true*, we have to go back through a second time to find the position of the element. It would be more efficient to use one loop to do both jobs. Write a private find helper method that works just like contains, except that it returns the index in the array where the element was found instead of *true*, and −1 instead of *false*. Call find, and if it indicates that the element was not found, you can return *false*; otherwise, you need to remove the element at that index. One way to remove the element would be to move all the following elements down one cell, but since we don't care about order (we’re building a *Set* after all), there's no need to do all that work. Instead, simply replace the element to be removed (you already know its index by now) with the last element in the elements array, decrement size, and return *true*.

Next, we need to consider the iterator method. This method needs to return an object of type Iterator <ElementType> that's somehow attached to this set; thus, we need a class to create such an object. Just like we did for the array-based list example from class, we’ll create an inner class for this purpose; that is, we'll make an *Iterator* class inside the <*FOO>ArraySet* class, just before the closing right brace character. Here's the shell of the code:

private class InnerIterator implements Iterator<ElementType> {

private int nextPosition;

public InnerIterator () {

nextPosition = 0;

}

public boolean hasNext() {

// your code here

}

public ElementType next() {

// your code here

}

public void remove() {

throw new UnsupportedOperationException();

}

}

The constructor resets nextPosition to *0*. The hasNext method returns *true* when there are still elements to be returned. The next method throws a NoSuchElementException if there is no next element; otherwise, it returns the element at nextPosition and increments the nextPosition variable. We don't choose to support element removal through the iterator, so its remove method throws the specified exception.

Back in the outer class, the iterator method should return a new instance of the InnerIterator class.

The last four methods to consider are the mathematical set operations intersection, difference, union and subset. None of these methods should mutate this set or the parameter set. For the first three methods, start by creating a new empty set that will contain the result such as

*<*FOO>Set<ElementType> result = new <FOO>ArraySet<ElementType>();   
  
except that you need to use your initials instead of *<FOO>*.

Note that since our class implements interface *<FOO>Set* which extends interface *ZHCollection* which, in turn, extends interface *Iterable*, you can (AND SHOULD) use the enhanced for-loop to iterate over the elements in a set.

For intersection, iterate through the elements of this set and add to the result the elements of this set that are also contained in otherSet. Method difference is very similar to intersection so you should be able to figure out the logic on your own*.* For union, you’ll need to iterate through both this set and otherSet adding all the elements of both; since the add method checks for duplicates, no element will be added twice. Finish these three methods by returning the result set. For subset, iterate through the potentialSubset, and for each of its elements, test whether this set contains the element. If you find an element in the potentialSubset not in this set, return *false*. If you exit the loop without finding such an element, return *true*.

Debug the code so that it compiles without errors.

***Part 3: Testing the class***

File *<FOO>SetTest.java* is a COMPLETE JUnit test class that allows you to test your class once it compiles. If you haven’t done so already, rename itwith the (capitalized) initials you are using for your structures (e.g., I would name it ***IMR****SetTest),*change the import <foo> structures.\*; statement to your structures package, and again change all instances of “<*FOO*>” to the (capitalized) initials you are using for your structures. When you have finished testing and believe your implementation is correct, show it to the lab instructor or the TA.

***Problem 3: Using your Set ADT implementation in the worker application***

Create a copy of your complete *WorkerManager* program from problem 1; call it *WorkerManagerV2*. Make all the necessary changes so that this new *WorkerManager* program uses the *<FOO>Set* interface and *<FOO>ArraySet* ADT in YOUR structures package from problem 2 instead of JCF’s (Java Collections Framework) *Set* interface and *HashSet* ADT. Please note that this will require an *import* statement in *WorkerManagerV2* to give it access to your structures package.