Lab 3 – Using and implementing sets

Continue working with your pair partner(s) on this lab.

Objectives:

- to use existing ADTs in the Java Collections Framework
- to construct and implement our own complete ADT
- to use existing software components to support building new software
- to create programs made up of multiple interacting Java classes
- to gain more insight into using arrays to build data structures

Problem 1 (of 3): Creating a worker application using sets

Follow instructions in this section to complete a number of programs designed to manage a set of different types of workers (*Set<Worker>*). Recall that your *worker* package from lab 2 sits inside folder *JavaPackages* on your home directory so it will be accessible from this lab folder provided you use an `import worker.*`

Part 1.1: Complete application class WorkerManager

Related class sample that you'll find handy: ***HashSetExample.zip***.

Keep file *WorkerManager.java* inside your *CS200/Labs/Lab03_Sets* folder [Meaning: DO NOT MOVE it to *JavaPackages*]. We will be saving *Worker* objects to an object file so make sure interface *Worker.java* in package *worker* extends `Serializable` from `import java.io.*`

In this lab, you will use one of Java's set ADT implementations called `HashSet`. The complete documentation is available here *https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/HashSet.html*; but we are mostly interested in the `Set` interface implemented by Java's `HashSet` class. Please access the following URL *https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Set.html* to study the methods available to you in this interface very carefully as we'll need to use them for the set *workers* instance variable in class *WorkerManager*. By the way, Java's `HashSet` is the main reason we overrode method `hashCode` in the *worker* package in lab 02. We will come back to this topic later in this course.

The *WorkerManager* program given to you inside your lab folder provides a good starting template for your application, but many of the pieces remain to be filled in. These places are all clearly marked with comments in the program. Start by carefully reading and fully understanding the provided incomplete *WorkerManager* class before moving to the steps below. Ask plenty of questions; after all, you can't complete/modify what you don't understand.

[1] Method *add* is complete, but it calls three helper methods: *newVolunteer*, *newHourlyEmployee* and *newSalariedEmployee* that are not complete. Complete these methods first as described in their Javadocs.

[2] Next, partially complete the *report* method by having it print a listing containing the names and worker types for ALL 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 will modify this method later.

At this point you should be able to compile and run the program as it stands. Add several different types of workers and report all the workers in the set. Make sure you have this much working before you go on.

[3] Once methods *add* and *report* are working, move on to *remove*. 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 need to break from the enhanced *for* loop (using a `break` statement) right after you remove the match from the set.

[4] Next implement the *save* and *load* methods; you will need to write or read the whole set as a single object to or from an object file, respectively. Note that Java's `HashSet` already implements interface `Serializable` which allows you to save it to an object file (as you did with lists in the last lab).

[5] Finish the *WorkerManager* by completing the *manage* and *report* methods as well as any other parts you did not get done so far. For the *report* method, use the *add* method as a template to prompt the user for the type of *Worker* 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). In addition to the name and *Worker* type, you'll need to report additional information applicable to different types of worker (such as monthly pay for any *Employee* and hours and hourly rate for *HourlyEmployees*).

Part 1.2 Creating three additional manager application programs

Class *WorkerManager* only manages properties common to all types of workers; specifics related to a *Volunteer*, *HourlyEmployee* and *SalariedEmployee* need to be managed separately by their own manager classes. As a result, in order to complete the *manage* method in class *WorkerManager*, you need to create three new Java classes: *VolunteerManager*, *HourlyEmployeeManager*, and *SalariedEmployeeManager*.

These classes 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 a single instance field for the *Worker* object it will manage [e.g., *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 types of specific management operations appropriate for each type of worker based on the classes themselves. There will be some overlap, but they will not generally be the same.

[1] *SalariedEmployeeManager* should allow us to set and view the monthly pay

[2] *VolunteerManager* and *HourlyEmployeeManager* should allow us to add, reset, and view current hours

[3] Additionally, class *HourlyEmployeeManager*, should allow us to set and view the hourly pay rate and view the monthly pay

Method *manage* in class *WorkerManager* simply creates the appropriate type of manager class (passing the current *Worker* object as a parameter) and then calls its corresponding *runManager* method similar to how the *main* method in *WorkManager* is currently written.

Problem 2 (of 3): Implementing your own ArraySet ADT

This problem is the first of several in which you will construct your own collection ADT by completing a class that implements an interface either provided to you in the lab folder or through the infamous *zhstructures* package.

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 will use your initials to name the package and its associated classes and interfaces. For example, your professor, ***I****mad* ***M R****ahal*, would name his package ***imr****structures* and the class we are constructing today, ***IMR****ArraySet*. Please note that names are case sensitive; by convention, package names are all lower case, while interface and class names start with an uppercase letter.

For the rest of this lab (and several to come), when names in the write-up or in given code begin with ***<FOO>*** or ***FOO***, replace the string with your uppercase 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.

Create a properly named folder in your *JavaPackages* directory for your new structures package (such as ***imr****structures*, in my case). You already have the corresponding class path directories entered into *DrJava* but if it complains about not finding *zhstructures* or things in your *JavaPackages* directory, seek assistance.

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

Move file *FOOSet.java* into your structures folder but keep the *FOOArraySetTest.java* in your lab 03 folder. Rename *FOOSet.java* with the (capitalized) initials you are using for your structures (e.g., I would name mine *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 new structures package.

Make the following changes to file *FOOSetTest.java*: rename it with the (capitalized) initials you are using for your structures (e.g., I would name mine *IMRSetTest*), change the `import <foo>structures.*` statement to your structures package, and change all instances of <*FOO*> to the (capitalized) initials you are using for your structures.

Next, create a new Java class called *FOOArraySet*, in the same package alongside the renamed *FOOSet.java*, but use the (capitalized) initials you are using for your structures instead of *FOO*. Do not forget to include an appropriate `package` statement at the top and to implement the renamed interface *FOOSet* (use an `implements` clause in the class header). You'll also need to import package `java.util.*` and 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 the renamed interface *<FOO>Set* to your class and include two constructors: one with no parameters and a second one 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 not do anything for now since we have not added any instance variables yet. Your class should compile without errors, although there may be warnings about unused parameters.

Now you are ready to start implementing the Set ADT. Add appropriate Javadoc comments for your class instance fields, constants, constructors as well as any additional methods not specified in the implemented interface, and use *{@inheritDoc}* for methods already there. Do this as you write your code not afterwards.

Observers: Please note that, after getting a copy from the driver, the observer will have to rename their package, classes and interfaces using their own initials.

Part 2.2: Implementing and testing class <FOO>ArraySet

Related class exercises that you'll find handy: ***ArrayListWithIterator.zip***, ***MyArrayStack.zip, MyArrayQueue.zip***

Name all fields and methods EXACTLY as shown here; otherwise, you're likely to encounter failed JUnit tests.

Let us 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 will use an idea similar to what we did for the *MyArrayList* class exercise so please keep the exercise solution handy during this lab (the version with iterators).

Click on the following link to access our *Algorithm Visualizations for CSCI-200: Data Structures* library webpage (compliments of your awesome professor!). Specifically, follow the link to *Lab03_Sets* *Set: Expandable Array Implementation* algorithm). Recall that you have complete control over the speed of the animation.

The goal of today's animation is to help you better understand the exact logic behind the two main operations in your ADT: ***add*** and ***remove***, before you start programming. You should spend ample time to understand the expected logic and note that only solutions adhering to this logic will be considered accurate. At a minimum, you should study the following cases and closely observe what happens (remember that as a computer scientist, you are always expected to make sure any program you produce is thoroughly tested for normal, abnormal, and boundary cases).

- ***add***: (1) add about 6 elements to the set;
  (2) try adding duplicates;
  (3) try adding beyond capacity;

- ***remove***: (1) remove elements from the set located at the (a) start of the array, (b) end, and (c) middle;
  (2) remove all the elements in the set;
  (3) trying removing elements that do not exist;

Keep referring to this visualization throughout the remainder of today's lab. Following our pair programming model, the observer should be in charge of the visualization on one computer while the driver writes code on the other. Switch roles often.

To implement this ADT, we will need two `private` instance variables—an `ElementType[]` array called `elements` and an `int` variable `size` initialized to *0*—along with a `public static final int` class variable `DEFAULT_INIT_LENGTH`, initialized to a reasonable power of 2 such as 1024.

Next, we will implement methods needed for our array set class implementation and test them as we go along. Take a close look at test class *FOOSetTest* inside your lab folder. This is a COMPLETE JUnit test class; if you have not done so already, rename it with the (capitalized) initials you are using for your structures, change all instances of *foo*/*<foo>* and *FOO*/*<FOO>* to the (lowercase and uppercase, respectively) initials you are using for your structures (do a find and replace). Take time to look at the test methods inside this class (at least read their names to understand what they do). The last few test methods test the structure of the interface and classes we're building here; some are likely not to pass until the very end so don't be frustrated.

[1] Consider method `init()` in the test class; since this method runs before every test, we need to make sure all methods and constructors called here are implemented first. The method creates test instances using both constructors (default as well as a one-parameter constructor) and calls method `add`. Thus, we will start by implementing these in our array set implementation class.

[2] The default constructor creates a set with a capacity (array length) of `DEFAULT_INIT_LENGTH`; the second constructor creates a set with a capacity (`initLength`) specified as a parameter by the user.

[3] The second constructor 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. Since the second constructor has a parameter, consider whether there are possible invalid values for the parameter, and if so, the constructor should throw an `IllegalArgumentException`. In this case, be sure to document the exception with an `@throws` tag in your Javadoc comments for the constructor.

[4] For method `add`, 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 is easiest to put it at the end of the block as shown in the visualization. Use the visualization to add a duplicate element to the set; it should fail. Thus, before adding a new element, we need to check if it is already there (using method `contains` which we will implement shortly). If the element is already in the set, the method should change nothing and return *false*.

Use the visualization to add beyond the capacity of the array; it should automatically double its size. Thus, once the `add` 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*.

[5] The `reallocate` method should be `private` since it is 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 is done copying `elements`, it stores the reference to the new array in the `elements` instance variable and returns.

[6] Next, implement the `isEmpty`, `size` and `contains` methods. These are straightforward and do not require separate testing, but as can be seen in the test class, are used to test constructors and more complex methods. 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.

[7] Once you have implemented the constructors and methods so far, take the time to test them by running your test class. You should expect to find a bunch of related errors; if so, go back to your implementation, and address the issues until resolved. Do not proceed before all *expected* tests pass but recall that some of the structure tests at the bottom are likely to continue to fail at this point.

[8] Next, use the visualization to study how `remove` behaves for elements that do not exist in the set as well as for ones that do. It is obvious that `remove` is a little trickier than `add`. First, 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 loop only once 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.

An inefficient way to remove the element would be to move all the following elements down one cell (like we did in the class exercise), but since we do not care about order (we are building a *Set* after all), there is no need to do all that work. Instead, simply replace the element to be removed with the last element in the `elements` array, decrement `size`, and return *true*. This can be seen in the visualization.

When done, run your test class to check that tests for the `remove` method are successful; in addition, make sure tests from before are still successful! After all, we always want to make progress without breaking anything completed earlier. If any test fails, go back to your implementation, and address the issues until resolved. Do not proceed before all expected tests pass.

[9] Next, we need to consider the `iterator` method. This method enables us to iterate over elements of this set using an enhanced for-loop (such as `for(ElementType e: ThisSet)` it will be very useful in implementing the remaining methods in your class.

Method `iterator` needs to return an object of type `Iterator <ElementType>` that is somehow attached to this set; we need a class to create such an object. Just like we did in the array list with iterators example from class last week, we will create an inner class for this purpose; that is, we will make an `InnerIterator` class inside ***FOOArraySet***, just before the closing right brace character. Back in the outer class, the `iterator` method should return a new instance of the `InnerIterator` class. You should closely follow example `ArrayListWithIterator`. Rerun your test class to pass the `iterator` method tests.

[10] The last four methods to consider are the common mathematical set operations `intersection`, `difference`, `union`, and `subset`. None of these methods should mutate (or change) this set, or the parameter set. For each of the first three methods, start by creating a new empty set that will contain the result such as below, only using your *Set* ADT instead.

```
<FOO>Set<ElementType> result = new <FOO>ArraySet<ElementType>()
```

Since our class includes an `iterator` method and implements interface *FOOSet* which extends interface *ZHCollection* which, in turn, extends interfaces `Serializable` and `Iterable`, you SHOULD use the enhanced for-loop to iterate over the elements in the set.

Implement one method at a time then rerun the test class to make sure you have succeeded before moving on to the next. Here are some helpful pointers:

- For `intersection`, iterate through the elements of this set adding to `result` any elements also contained in `otherSet`. Return the `result` set when done.
- `difference` is very similar to `intersection` so you are on your own.
- `union` is also very similar, but you will need to iterate over elements in both sets knowing that your `add` method will restrict adding duplicates to `result`.
- `subset` is more complicated. You need to iterate over parameter `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*.

Problem 3 (of 3): Using your Set ADT implementation in the worker application

Create a copy of your complete *WorkerManager* program from *Problem 1*; call it *WorkerManager**V2***. Make all the necessary changes so that this new *WorkerManagerV2* program uses the set ADT in YOUR structures package (i.e., *FOOSet* and *FOOArraySet* from *Problem 2*) instead of Java's `Set` ADT and its `HashSet` implementation. Note that this will require an additional *import* statement in *WorkerManagerV2* to give it access to your structures package. Other changes may be required as well.

Welcome to CSCI-200, for real! 😊