LAS3006 – Worksheet 1 – Lambda Expressions

1. **Integer Array Manipulation using Lambda Expressions.**
   a) Write a method which fills an **`Integer`** array of a pre-defined size with **random positive integers**. Use **`Arrays.setAll()`** to invoke your random generation code. **Do not iterate through the array yourself.**

   b) If it is not already in a separate method, put the random generation code in a separate function. Call it from **`Arrays.setAll()`** as a **method reference**.

   c) Create a **`Comparator<Integer>`**, that puts the **shortest** integer first, in terms of number of digits (you can just convert the integer to a string first). If 2 integers have the same number of digits, sort them in **descending order**. Use a **lambda expression** to create it inline.

   d) Sort the array created in step 1, (using **`Arrays.sort()`**) and pass the Comparator you created in step 3.

   e) Print out the contents of the array. Try it with an array of size 100.

2. **Filtered list using Lambda Expressions.**

In this exercise you are going to create a special **`List<E>`** implementation which only accepts elements that satisfy certain criteria. These criteria will be specified at runtime when instantiating the list, by passing a lambda expression at the constructor.

   a) Create a new class named **`FilteredList<E>`** that extends **`LinkedList<E>`**. As its constructor argument, the **`FilteredList<E>`** must take a functional interface whose method takes an element of generic type **`E`**, and returns true if the element should be accepted by the list, and false otherwise.

   Hint: Java 8 provides a ready-made functional interface for such needs called **`Predicate<E>`**.

   b) Override the **`add(E)`**, **`add(int, E)`**, **`addFirst(E)`** and **`addLast(E)`** so that prior to adding elements, they are tested with the lambda expression passed as a constructor of the **`FilteredList<E>`**. If the element does not pass the test, these methods must throw an **`IllegalArgumentException`** (with the appropriate message indicating the element).

   c) Override **`push(E)`** and **`offerFirst(E)`** as aliases of **`addFirst(E)`**.Override **`offer(E)`** and **`offerLast(E)`** as aliases of **`addLast(E)`**.

   d) Override **`addAll(Collection<E>)`** and **`addAll(int, Collection<E>)`** such that all the elements passed in the **`Collection<E>`** are tested **prior to adding any element**. (That is all the passed elements must be valid or none are added.)

Hint: You might want to encapsulate the code that tests the element in a separate method so that it is called by all the other methods, and can also be used by **`Collection.forEach()`**.

Test it out with a **`FilteredList<Integer>`**, and make it only accept odd numbers.

3. Advanced List Filtering using Lambda Expressions.

In this exercise you are going to create a more advanced List<E> implementation which not only accepts elements that satisfy certain criteria, but also offers the possibility for these criteria to be based on the existent elements of the list, and also to modify the element being added to the list.

a) Create a new class named **`AdvancedFilteredList<E>`** that extends **`LinkedList<E>`**. As its constructor argument, the **`FilteredList<E>`** must take a functional interface whose method takes a **`List<E>`**, representing the current contents of the list, the new element of type **`E`**, and returns **`E`**, to be added to the list (which could be a different value). The method should also throw **`IllegalArgumentException`**, in case nothing should be added to the list.

Hint: You can call this functional interface **`AdvancedListFilter<E>`**, having one method with signature **`E filter(List<E>, E)`**. Do not forget the **`@FunctionalInterface`**.

b) Override the **`add(E)`**, **`add(int, E)`**, **`addFirst(E)`** and **`addLast(E)`** so that prior to adding elements, they are tested with the lambda expression passed as a constructor of the **`AdvancedFilteredList<E>`**. (In this case the functional interface will automatically throw an **`IllegalArgumentException`** by itself.)

Note: Remember that the **`AdvancedListFilter`** could modify the value that was originally passed. Make sure that you add the return of the **`filter()`** method to the list.

c) Override **`push(E)`** and **`offerFirst(E)`** as aliases of **`addFirst(E)`**.Override **`offer(E)`** and **`offerLast(E)`** as aliases of **`addLast(E)`**.

e) Override **`addAll(Collection<E>)`** and **`addAll(int, Collection<E>)`** such that all the elements passed in the **`Collection<E>`** are tested as they are added. If any element fails the test it is skipped with an error message output to **`System.out`**, but the rest of the elements should be added.

`Hint:` You can still use a **`forEach()`** for **`addAll(Collection<E>)`**, since you can catch the **`IllegalArgumentException`** within the lambda expression. For **`addAll(int, Collection<E>)`** this is a bit more tricky, since you have to keep track of which index you are at, and thus the variable is not ***effectively final***. (Use a normal for-each loop for this exercise, we will see other mechanisms to propagate counters and other accumulated information in future lectures).

Test it out with an **`AdvancedFilteredList<Integer>`**, and make it only accept numbers which do not have a factor already in the list. If the number passed is negative, first convert it to a positive number before testing it and adding it to the list. (Do all this logic within the lambda expression). Do not forget to catch **`IllegalArgumentException`** when adding elements one by one.

For example: Adding the following sequence of integers:

81, 9, 27, -5, 14, 15, 3, -6

Should result in the list containing:

81, 9, 5, 14, 3