Task

Introduction

In this exercise you create a variety of containers that you will need for the next assignment. It is therefore important to write good code that is easy to read, because you have to come back to and re-use the code you write along. We need some different varieties of link lists, so we're creating more classes that inherit from each other and implement different interfaces.

When you solve the task, it is particularly important that you think about what the different conditions a container can have. How should the condition be for an empty link list, and how this changes when you insert an item? Will the condition be as before when removing an item from the list? Such questions should you ask yourself along the way.

If you want more flexible classes, they can of course have several methods in addition to those required by the assignment text. Note that it is also up to you how you choose to structure references between nodes - you can choose whether your list should be single-linked (all nodes have a next-pointer) or double-linked (nodes also have a pointer to the previous node in the list) .

About unit testing of components

To test your classes, you should download and run a number of given test classes that you can be find (attached in the email “Testclasses”).

You will test the classes you create along the way against both these unit tests. At the same time you should also do own tests, for example by using the classes you wrote in the last task. The unit tests should be attached to the assignment, and it is therefore *important* that the names of the required methods follows the text. *Also make sure* that the downloaded files are located in *same folder as* your classes.

Part A: Class Hierarchy

Draw a class hierarchy (including interface) to the different classes to be written. It is important that you read the entire assignment before solves this part!

Part B: Link List

In this task, we base ourselves on the interface List <T>. The interface looks like this:

```
interface List < T > {
        public int size ();
        public void add( int pos , T x );
        public void add( T x );
        public void set ( int pos, T x );
        public T get( int pos );
        public T remove(int pos );
        public T remove();
}
```

In this task we will create link lists using a similar interface.

B1: Write class *LinkList<T>* that implements *List<T>.* We should be able to easily insert items at the end of the list and take out from the start so that the first item that was inserted is also the first to be taken out. In this way, the list is used as a queue (First in, first out). Method *add(T x)* should therefore insert an item at the end of the list, while *remove*() will remove and return the item to the beginning of the list.

In addition, there are overloaded methods to set, add and remove at given pitches:

- method *set(int pos, T x)* insert element in a given position and overwrite what was there already.
- method *add(int pos, T x)* will add a new item to the list and push the next item one notch further behind.
- method *remove (int pos)* will remove on given index in the list.

Finally, you must also implement methods *size()* and *get(int pos) ,* the latter retrieves an item (without removing it from the list) at specified index (remember to count from index 0 and up).

B2: When we work with indexes, we can face mistakes if we are trying to reach an index that does not exist. Valid indexes in the list will be that we are accustomed to in an array or an ArrayList, ie from 0 and up to, but not including, the size of the list. To account for any errors, we will use this custom exception class:

```
class InvalidListIndex extends RuntimeException                    {
      InvalidListIndex ( int Index )                {
            super ( " Invalid Index " + Index );
```

Download and save this exception class with java files, and make sure it is discarded if we try to reach an invalid index (or if we try to remove something from an empty list - in that case we will throw the exception with index -1).

B2: Make sure your list is going through the relevant tests (“TestLinkLint.java”, sent on email) before moving on to Part C.

Part C: Stack

A stack is a list that works a bit differently than a regular link-list. When you add an item in, it will be the first to be retrieved.

C1: Write class Stack <T>. The class will inherit from LinkList <T>, but must also have methods *Addon(T x)* and *removeoff()* .These methods will respectively add and remove elements from the end of the list so that the last item entered is the first to be removed (Last in, first out). Note: It is expected here that you use the methods that are inherited from LinkList <T>.

C2: Make sure your list is going through the relevant tests (TestStack.java, sent on email) before moving on to Part D.

Part D: Sorted Link List

D1: Write class SortedLinkList <T extends Comparable <T>>. This list also inherits from LinkList <T>, but we want the list to be sorted and therefore require that elements inserted should be comparable. Call method add(T x)should insert elements in sorted order (from smallest to largest), and when we use *remove()*- method (without parameters) the largest element should be removed.

D2: You should limit the opportunities for inserting elements at an arbitrary position, so that the list stays sorted. This can be done by allowing SortedLinkList overwrite the methods set (int pos, T x) and add(int pos , T x). The new implementations will not insert element, but instead just throw an exception that exists in Java from before:
UnsupportedOperationException (this does not need to be imported).

D3: Make sure your list is going through the relevant tests (TestSortedLinkList.java, sent on email).

Summary

You should submit the following:

- Drawing of the class hierarchy (as image file or .pdf)
- Classes LenkList.java, Stack.java and SortedLinkList.java
- Attached interfaces and exceptions (List <T> and InvalidListIndex).