OBJECT ORIENTED PROGRAMMING – ADTS

Unassessed Tutorial 1: *ADTs*, *Lists* and *Linked List*

The aims of this tutorial are:

- To practice writing an ADT for a given problem.
- To implement some of the access procedures of an ADT List.
- To practice writing examples of client methods that use List's access procedures.

1. a) Define and implement an ADT that represents a bank account. The data of the ADT should include the customer name, the account number, account balance and the allowed overdraft. The creation of the ADT should set the data to client-supplied values. Include operations for deposit and withdrawal of a given amount, and for viewing the current balance.
   - `void viewBalance()`
     // post: print the current balance.
   - `void deposit(double amount)`
     // post: add the given amount to the current balance.
   - `void withdrawal(double amount)throws withdrawalException`
     // post: subtract the given amount from the current balance if it does not go below the
     // post: allowed overdraft. Throw an exception otherwise.

   b) Write a class `ClientAccount` that creates an account, view the current balance at creation, performs a deposit, a withdrawal and view the new balance. The following output is produced:

```
The current balance is 1200.0
The current balance is 2434.0
Exception in thread "main" withdrawalException: Not enough funds
        at Account.withdrawal(Account.java:23)
        at ClientAccount.main(ClientAccount.java:8)
```

2. Consider the class `ArrayBasedList<T>` given in Slide 7 of Unit 2, and assume the attribute `maxList` to be a variable. Implement the method `add(int givenPos, T newItem)` (declared below) that uses a dynamic expansion of `elems`. Provide also the implementation of any auxiliary procedure you may want to introduce:

```
public void add(int givenPos, T newItem) throws
                              ListIndexOutOfBoundsException
```

3. Consider the generic interface `List<T>` given in Slide 2 of Unit2. Implement the following methods of a client program:

   (a) `public void swap(List<String> myList, int pos1, int pos2)`
   //pre: `myList` is not empty, and `pos1` and `pos2` are within the size of `myList`.
   //post: swap the elements at positions `pos1` and `pos2`.

   (b) `public void copy(List<String> listFrom, listTo)`
   //pre: `listTo` is empty
   //post: Copy all elements of `listFrom` to `listTo` leaving `listFrom` unchanged.

   (c) `public void reverse(List<String> myList)`
   //post: Reverse the order position of the elements in `myList`
   (**Note**: Implement this method without creating and using auxiliary list objects)

   (d) `public void doubleFirst(String item1, item2, List<String> myList)`
   //post: inserts `item1` twice before each occurrence of `item2`.

   (e) `public List<T> alternate(List<T> myList)`
   //post: returns a linked list containing the first, third, fifth, … elements of `myList`.
   //post: `myList` is left unchanged.

4. Consider the generic interface `List<T>` given in Slide 2 of Unit2, and assume the existence of a public class `Node<T>` as defined in Unit 2. Implement the following access procedures of the generic class `LinkedBasedList<T>`:
   (Hint: use the auxiliary method `Node<T> getNodeAt(int givenPos)` given in Slide 22 of Unit 2.)

   (a) `public T get(int givenPos)throws ListIndexOutofBoundsException`

   (b) `public void add(int givenPos, T newItem) throws ListIndexOutofBoundsException`

   (c) `public void remove(int givenPos)throws ListIndexOutofBoundsException`

5. Repeat Question 4 above but assuming the class `Node<T>` of Question 4 to be now an inner class of our class `LinkedBasedList<T>`.

6. Consider an ADT `Polynomial` (in a single variable x, and powers not negative integers) whose access procedures include the following:
   - `int degree()`
     // post: return the degree of the polynomial.
   - `int coefficient(int power)`
     // post: return the coefficient of the x^{power} term.
   - `void addTerm(int power, int coefficient)`
     // post: add a new term with given `power` and `coefficient`, if `coefficient` is
     // post: not zero. If a term with the same `power` exists, it adds the `coefficient`.
   - `void changeCoefficient(int newCoeff, int power)`
     /// post: replaces the coefficient of the existing x^{power} term with the `newCoeff`, if
     // post: different from zero, or delete the term if `newCoeff` is zero. If x^{power} term
     // post: does not exist adds a new term with the given `newCoeff` and `power`, only
     // post: if `newCoeff` is different from zero.

   (a) Assume the existence of an ADT List<T>. Give a dynamic implementation of the ADT `Polynomial` that reflects the UML diagram given in Figure 1 (see below).

   (b) Using the ADT Polynomial give the implementation of the method

```
public PolynomialADT SumPolynomials(Polynomial firstPoly, secondPol)
```

   //post: sum the given two polynomials and return the resulting polynomial.

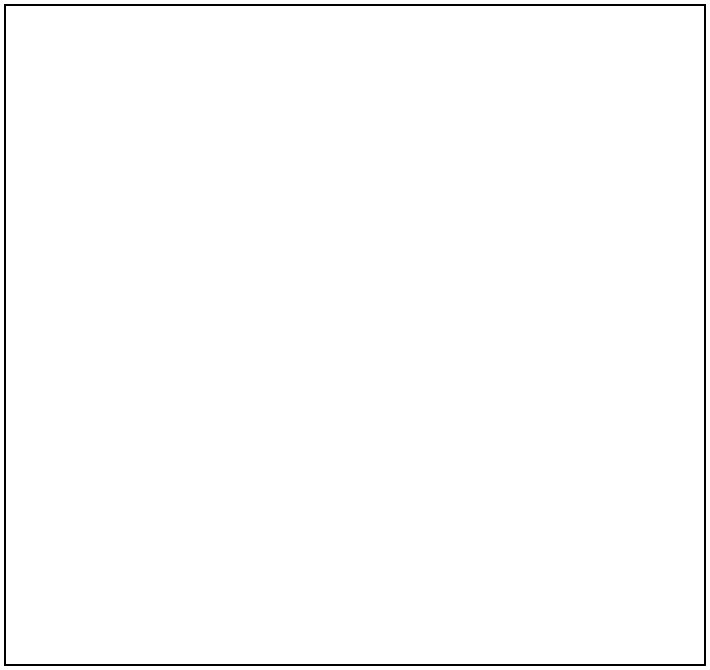

Figure 1