Lab 9. Recursion

Theme. In this lab, you will:
- use recursion to implement some operations of a linked list
- use a linked list to keep a ranked list of email contacts
- explore when is suitable for recursion to be used

Key concepts: recursion, indexed lists, linked lists

Required file(s): `lab9.zip`

1 Getting started...

1. In a web browser, download the archive `lab9.zip` from Blackboard and extract its contents into your default `eclipse` workspace for this module.
2. Start up `eclipse`.
3. Making use of the contents of your extracted archive, create a new Java project named **addressbook** in your `eclipse` workspace using the contents of your extracted archive.

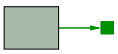
2 Working with Recursion

Recursion is a programming technique in which a method can *call itself* to solve a problem. When used appropriately, recursion can make a program much **shorter** (but not necessarily more efficient) than its iteration counterpart. It can make the implementation much simpler and more elegant. For example, recursion is a natural choice for implementing various operations on a tree data structure as it does not require the knowledge of the size of the tree (i.e. the number of nodes that are in the tree).

On the other hand, in Java, recursion is *not* the answer to *all* software problems. It is important to know when is beneficial to use recursion to solve a problem. This knowledge requires experience. Hence, the first step to gain such an experience is to try out implementing various software solutions using recursion and see how easy or difficult they are. This Lab's exercises are designed to help you gain such experience.

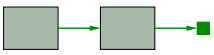
Don't forget that a recursive method should always have two parts:

- base case, e.g. the base case for a linear linked structure is:



  I.e. a node that has a `null` follower.

- recursive step, e.g. the recursive step for a linear linked structure is:



  I.e. a node that has a non-`null` follower.

The archive `lab9.zip` contains one package `addressBook` which has *one* interface and *four* classes:

- `List`: a generic Java interface to specify the ADT list. Note that not all typical operations of a list have been included in this interface.

> **Hint**: Note that interface `List` is different from the one in JCF.

- `LinkedList`: a generic class for modelling a linked list collection using a linked structure. This class implements interface `List`. Note that not all typical operations of a linked list have been included in this class. A linked list keeps track of the *first* node within the list *only*.

> **Hint**: Note that class `LinkedList` is different from the one defined in JCF.

- `LinkedNode`: a generic class for modelling a *node* within a linear linked structure. A `LinkedNode` object keeps the object reference to an element within the linear linked structure. It contains operations for adding another `LinkedNode` object as its successor, removing a `LinkedNode` object that is its successor, and so on.

> **Hint**: A "successor" of a `LinkedNode` object does *not* need to be the node that immediately follows this `LinkedNode` object. It could be a node that appears further down the linear linked structure.

- `EmailContact`: a class for modelling an email address record of a contact (person).
- `AddressBook`: a class for modelling an email address book which keeps email address records according to their importance to the address book owner. If the email contact is very important, it is kept as the first entry within the address book. If the contact is the least important, it would be kept as the last entry in the address book. The email address records are kept in a linked list, i.e. an *indexed list*.

Your task is to complete the given partial implementation of classes `LinkedNode` and `LinkedList` by following the given instructions.

> **Hint**: The rough locations where you are expected to add your Java code and relevant hints for accomplishing the tasks have been marked throughout the given Java programs. Look out for **block comments** that include a sequence of *four* exclamation marks, i.e.:
>
> ```
> /* !!!!                    */
> ```
>
> The locations in the given Java code where you are expected to pay particular attention have also been annotated. Look out for **block comments** with a sequence of *four* plus signs, i.e.:
>
> ```
> /* ++++                    */
> ```

Your Tasks

1. Complete the implementation for the method `size` in `LinkedNode` using *recursion*. This method should return the number of nodes that are linked together, counting from the `LinkedNode` object itself.
2. Complete the implementation for the method `toString` in `LinkedNode` using *recursion*. This method should return a `String` representation of all elements that are linked together, starting from the `LinkedNode` object itself.
3. Complete the implementation for the method `getElementAt` in `LinkedNode` using *recursion*. This method should return the element at a specified position within the list that is headed by the `LinkedNode` object.
4. Complete the implementation for the method `size` in `LinkedList`. This method should simply use the `size` method of its instance variable `first` to return the size of the linked list. There is no need for any use of iteration or recursion in this implementation.
5. Complete the implementation for the method `getElementAt` in `LinkedList`. This method should return the element at a specified position within the list by calling the `getElementAt` method of its instance variable `first`. There is no need for any use of iteration or recursion in this implementation.
6. In class `AddressBook`, read the block comments marked with:

   ```
   /* ++++            */
   ```

   This will help you familiarise with how this class works before conducting the system testing.

3 Testing

Now, test your implementation to see if it meets the above requirements. To test your `LinkedList` implementation, you may use the given `main` method. A successful implementation of this class should produce the following output:

```
Creating an empty linked list...
The list contains:

Add 3 numbers to the linked list...
The list contains:
1
The list contains:
1
2
The list contains:
1
3
2
The list has 3 element(s).
Get an existing element...
3
Try to get a non-existing element...
java.util.NoSuchElementException
```

```
Remove the 2nd element...
3
The list contains:
1
2
Remove the last element...
2
The list contains:
1
Remove the first element...
1
The list contains:

The list has 0 element(s).
```

To test the complete implementation, you should use the given `main(String[])` method in the class `AddressBook`. If your implementation is correct, the output of the program should be:

```
Add Amy, Brian and Cathy to the address book...
Amy amy@home.uk
Brian brian@safari.com
Cathy cath@cathaypacific.hk

Add Sandy to the 3rd position...
Amy amy@home.uk
Brian brian@safari.com
Sandy rock@yahoo.com
Cathy cath@cathaypacific.hk

Add Philip to the 1st position...
Philip phil@yahoo.com
Amy amy@home.uk
Brian brian@safari.com
Sandy rock@yahoo.com
Cathy cath@cathaypacific.hk

Move the contact at the 1st position to the 4th position...
Amy amy@home.uk
Brian brian@safari.com
Sandy rock@yahoo.com
Philip phil@yahoo.com
Cathy cath@cathaypacific.hk
```

4 More Challenging Tasks

1. Class `AddressBook` assumes that the client will always supply appropriate ranking for an entry that is to be added or removed. Hence, it does not handle the cases when the supplied ranking exceeds the size of the email address list. Modify the methods `reRankEntry` and `addEntry` in class `AddressBook` in the following ways:

   - For `addEntry`:

- ◇ When a supplied ranking is less than 1, throw an exception with an appropriate message.
- ◇ When a supplied ranking is beyond the size of the list even after the new element is added, add the element to the end of the list.

- For `reRankEntry`:
  - ◇ When a supplied supposedly existing ranking does not exist in the list, throw an exception with an appropriate message.
  - ◇ When a supplied new ranking is beyond the size of the list, add the element to the end of the list.

2. Make `LinkedNode` become a `private` *inner class* in `LinkedList`.

   Class `LinkedNode` should be used by the class `LinkedList` *only*. To avoid a client class to "accidentally" create a `LinkedNode` object and hence messing up the logic of the underlying computer application, it is better to *hide* the class `LinkedNode` from other classes, except `LinkedList`. This cannot be achieved by simply changing the *visibility* (i.e. by using the Java keywords `private`, `protected`, etc) of this class. Thankfully, the concept of *inner class* in Java enables us to do so.

   We looked at the use of an inner class in Unit 10 when implementing an iterator for class `DoubleList`. If you have forgotten about how to use inner classes in Java, you will find a good explanation of inner classes in the section on "Nested Types" (pp.140–156) in the book "*Java in a Nutshell*" 5th Edition by David Flanagan. See also "*The Java Tutorials*" on Nested Classes for details: `http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html`

   Making `LinkedNode` become a `private` *inner class* in `LinkedList` hides it from other classes (except the enclosing class `LinkedList`). The inner class has access to the fields and methods of the class it is nested in (including the `private` ones). Also the outer class has access to `public` and `protected` fields and methods of the inner class. If, instead, the inner class is `protected`, it is also visible to classes in the same package.

   Making a class an inner class enforces encapsulation and hence makes the design better.

   Classes `LinearNode` and `DoubleNode` from Units 5 & 10 could (probably should) have been inner classes of the `LinkedList` and `DoubleList` classes respectively.

5 The Ultimate Challenge: Questions to think about...

1. In class `LinkedNode`, read the block comments in the method `addAt`. Try to understand how this method works. Can you tell why this method needs to return a `LinkedNode` object?
2. The method `removeAt` in class `LinkedNode` *cannot* be implemented easily using recursion. The problem lies in the number of values that need to be returned at the end of its execution. Can you describe what exactly this problem is?
3. Would it be easier to implement `removeAt` recursively in a class `DoubleNode` with both `next` and `previous` links?