Linked Lists

A **linked list** is a collection of objects linked together by references from a object to another object. By convention these objects are names as **nodes**. So the basic linked list consists of nodes where each node contains one or more data fields AND a reference to the next node. The last node contains a **null** reference. Here is an example

```
node              node             node
_____________    ______________   ______________
| data | ref |   | data | ref |   | data |      |
|_____|_____|-->|______|_____|-->|______|_null_|
    ^          ^
   head        
```

Let us design the immutable class **Node** (see the correspondent code example *Node.java*). A node is sometimes called a **self-referential** object, since it contains an instance variable that refers to an object of the same class.

**Understanding the Node class constructors:**

The statement

```
Node P = new Node("Pitt");
```

creates an object referenced by `P`. The node stores the string "Pitt" and null reference. The statement

```
Node Q = new Node("Paris", P);
```

creates an object referenced by `Q` which stores the string "Paris" and the reference to node `P`, thereby connecting node `P` to node `Q`.

```
| Paris | o--|--->|Pitt |null |
|_______|     |_____|_____|
^       ^
|       |
Q       P
```

The above two statements can be written in one

```
Node Q = new Node("Paris", new Node("Pitt"));
```

**We use getNext() and getData() to access a reference and a data stored in the node,** so `Q.getNext()` references node `P`, and `Q.getData()` references the string "Pitt".
Example.

Let us assume the following list

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;1&quot;</td>
<td>next</td>
<td>&quot;2&quot;</td>
<td>next</td>
<td>&quot;3&quot;</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>___</td>
<td>___</td>
<td>___</td>
<td>___</td>
<td>___</td>
<td>___</td>
<td></td>
</tr>
</tbody>
</table>

and explain the effect of each fragment below. The list is restored to its initial state before each line executes

1. \( P = P \text{.getNext()}; \)

   It moves the reference \( P \) to the second node. The side effect of this is that the first node will be garbage collected.

2. \( P = P \text{.getNext().getNext()}; \)

   It moves the reference \( P \) to the third node. The side effect of this is that the first two nodes will be garbage collected.

3. \( P \text{.setNext}(P \text{.getNext().getNext()}); \)

   This makes the first node point to the last node. The side effect is that the second node will be destroyed.

4. \( P \text{.getNext().getNext().setNext}(P); \)

   This makes a circular linked list.

Functional Operations:

In a well-designed list data structure, you should be able to manipulate its elements without knowing anything about its data. Applying this to a linked list; you should be able to operate with a linked list without knowing anything about data stored in each node.

Basic/fundamental operations, which can be performed over linked lists:

1. insert a new element (prepend, append)
2. list traversal
3. delete an element
4. invert a list

See the code example \texttt{LinkedList.java}
Cloning Linked Lists:

Evaluate this code

```java
LinkedList list = new LinkedList();
list.prepend(new Node("first"));
list.append(new Node("last"));
LinkedList list2 = list;
```

The reference `list2` is an alias, it points to the same object. This means that once we change `list`, these changes occur in `list2` as well. In many cases it will be quite useful to have a full copy of an object. To obtain such a copy, we need to duplicate all nodes in the list. For this purpose, the `Object` class provides the method `clone()`, which we will override in the `LinkedList` class:

```java
public Object clone() throws java.lang.CloneNotSupportedException {
    LinkedList twin = new LinkedList();
    if(head == null) return twin;

    Node tmp = head;
    while(tmp != null) {
        twin.append(tmp.getData());
        tmp = tmp.getNext();
    }

    return twin;
}
```

`append()` is a quite expensive operation, because you need to traverse the whole list each time you append. There are alternative ways to implement the `clone()` method – either using an auxiliary reference (that points to the last element being appended) or using the `Stack` class.

Copy constructor:

This idea is alternative to clone. Instead of overriding `clone` and dealing with `CloneNotSupportedException`, we can simply implement a constructor that is responsible for creating a copy of the object of the same type:

```java
public LinkedList(LinkedList list) {
    Node tmp = list.head;
    while(tmp != null) {
        append(tmp.getData());
        tmp = tmp.getNext();
    }
}
```