BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
First Semester, 2017-17
Object-Oriented Programming (CS F213)
LAB-7 (Introduction to Collections)

AGENDA

1. **Anonymous inner classes**
2. **List, Arraylist, Iterator, ListIterator and Linked Lists**

1. Anonymous Inner Classes

In some cases you might find yourself writing an inner class and then using that class in just a single line of your program. For cases like this you have the option of using an anonymous inner class. An Anonymous class is created with a variation of the 'new' operator that has the form

```
new superclass-or-interface ( parameter-list ) {
methods-and-variables
}
```

This constructor defines a new class, without giving it a name, and it simultaneously creates an object that belongs to that class. This form of new operator can be anywhere where general new could be used. The intention of this expression is to create : a new object belonging to a class that is the same as superclass-or-interface but with extra methods-and-variables added. The effect is to create a uniquely customized object, just as the point in the program where you need it.

Note: If an interface is used as the base, the anonymous class must implement the interface by defining all methods that are declared in the interface and the parameter-list must be empty

1.1 Example:

```
public class Movie {
     //Declaring an interface inside the class ..
     interface Bookable {
         public void printTicket();
         public void giveTicket(String movie);
     }
```

```
   public void BookTheTicket() {
        // Writing an InnerClass that implements the above
        interface
        class EnglishMovie implements Bookable {
           String name ;

           public void printTicket() {
             giveTicket("BlindDate");
           }

           public void giveTicket(String movie) {
             name = movie;
             System.out.println("You have booked for the movie "
                                + name);
           }
        }

        // creating an object for the inner class ..
        Bookable hollywood = new EnglishMovie();

        // anonymous innerclass which is basing the interface..
        Bookable hindiMovie = new Bookable() {
           public void printTicket() {
               giveTicket("Bachna Ae Haseeno");
           }
           public void giveTicket(String movie) {
             String name = movie;
             System.out.println("You have booked for the movie "
                        + name);
           }
        };

        hollywood.giveTicket();
        hindiMovie.giveTicket();
    }

    public static void main(String[] args) {
        Movie easyMovie = new Movie();
        easyMovie.BookTheTicket();
    }
}
```

In the above example the EnglishMovie is inner class and for the instance hindiMovie we created Anonymous inner class .

1.2 Exercise:

```
public class Anonymous {
     public Circle getCircle(int radius) {
       // Write one line statement that returns object of Circle

     }

     public static void main(String[] args) {
          Anonymous p = new Anonymous();
          Circle w = p.getCircle(10);
          // The output here should give correct value of area
          // of the circle.
          System.out.println(w.area());
     }
}

class Circle {
     private int rad;
     public Circle(int radius) { rad = radius; }
     public double area() { return rad*rad; }
}
```

2. List Interface

A List is an ordered Collection (sometimes called a *sequence*). The user of this interface has precise control over where in the list each element is inserted. In addition to the operations inherited from Collection, the List interface includes operations for the following:

- Positional access — manipulates elements based on their numerical position in the list
- Search — searches for a specified object in the list and returns its numerical position
- Iteration — extends Iterator semantics to take advantage of the list's sequential nature
- Range-view — performs arbitrary *range operations* on the list.

There are two implementations:

- LinkedList gives faster insertions and deletions
- ArrayList gives faster random access

3. ArrayList

It is resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null.

ArrayList supports dynamic arrays that can grow as needed unlike standard Java arrays which are of a fixed length.

In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list.

3.1 Example: Following example shows the use of ArrayList and its methods.

```
import java.util.ArrayList;

public class ArrayListDemo {
     public static void main(String[] args) {
          //Creating a new ArrayList
          ArrayList arlTest = new ArrayList();
          //Size of arrayList
          System.out.println("Size of ArrayList at creation: "
+arlTest.size());
          //Lets add some elements to it
          arlTest.add("B");
          arlTest.add("I");
          arlTest.add("T");
          arlTest.add("S");
          //Recheck the size after adding elements
          System.out.println("Size of ArrayList after adding
elements: "+arlTest.size());
          //Display all contents of ArrayList
          System.out.println("List of all elements: " +
arlTest);
          //Remove some elements from the list
          arlTest.remove("B");
          System.out.println("See contents after removing one
element: " + arlTest);
          //Remove element by index
          arlTest.remove(2);
          System.out.println("See contents after removing
element by index: " + arlTest);
          //Check size after removing elements
          System.out.println("Size of arrayList after removing
elements: " + arlTest.size());
          System.out.println("List of all elements after
removing elements: " + arlTest);
```

```
            //Check if the list contains "T"
            System.out.println(arlTest.contains("T"));
      }
}
```

Run the above code and observe the output.

4. Iterator

Each of the collection classes provides an iterator( ) method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time. In general, to use an iterator to cycle through the contents of a collection, follow these steps:

1. Obtain an iterator to the start of the collection by calling the collection's iterator( )
2. Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns true.
3. Within the loop, obtain each element by calling next( ).

5. ListIterator

An iterator for lists that allows the programmer to traverse the list in either direction, modifies the list during iteration, and obtains the iterator's current position in the list. A ListIterator has no current element; its *cursor position* always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next().

An iterator for a list of length n has n+1 possible cursor positions, as illustrated by the carets (^) below:

```
                    Element(0)   Element(1)   Element(2) ...  Element(n-1)

cursor positions:^           ^            ^            ^                ^
```

5.1 Example: Iterator and ListIterator

(1) Student.java

```
public class Student {
      private String name;
      private String gender;
      private int age;
```

```
	public Student(String name,String gender,int age){
	this.name=name;
	this.gender=gender;
	this.age=age;
	}

	public String getName(){
		return name;
	}
	public String getGender(){
		return gender;
	}

	public void setName(String name){
		this.name=name;
	}
	public String toString(){
		return name+"   "+gender+" "+age;
	}
}
```

(2) TestStudentList.java

```
// Demonstrate Iterator and ListIterator
import java.util.*;

public class TestStudentList {
	public static void main(String args[]) {
		// create an array list

		ArrayList studentList = new ArrayList();
		// add elements to the array list
		studentList.add(new Student("Ramesh","Male",18));
		studentList.add(new Student("Reeta","Female",19));
		studentList.add(new Student("Seema","Female",18));
		studentList.add(new Student("Suresh","Male",20));

		System.out.println("Original contents of
studentList:");
		Iterator itr = studentList.iterator();
		while(itr.hasNext()) {

		    Object element = itr.next();
		    System.out.print(element +"\n");

		}
```

```
            System.out.println();

            // modify objects being iterated
            ListIterator litr = studentList.listIterator();
            while(litr.hasNext()) {

                Student element = (Student)litr.next();
                if(element.getGender().equals("Male")){
                element.setName("Mr."+element.getName());
                }
                else{
                 element.setName("Miss."+element.getName());
                }
                litr.set(element);
            }
            System.out.println("Modified contents of studentList:
");
            itr = studentList.iterator();
            while(itr.hasNext()) {

                Object element = itr.next();
                System.out.print(element + "\n");

            }
            System.out.println();
            // now, display the list backwards
            System.out.println("Modified list backwards: ");
            while(litr.hasPrevious()) {

                Object element = litr.previous();
                System.out.print(element + "\n");

            }
            System.out.println();

     }

}
```

Exercise:
The L&L Bank can handle up to 30 customers who have savings accounts. Design and implement a program that manages the accounts. Keep track of key information and allow each customer to make deposits and withdrawals. Produce appropriate error messages for invalid transactions. Do this practice problem using ArrayList and Iterator.

(A) Create an Account class that tracks individual customer information.

```
public class Account {
  private long acctNumber;
  private double balance;
  private String name;

/*Complete the Account class by adding proper
constructor,accessor method
 and mutator method as required.Override toSring() method to
display account details.
*/
  //Write your code here

}
```

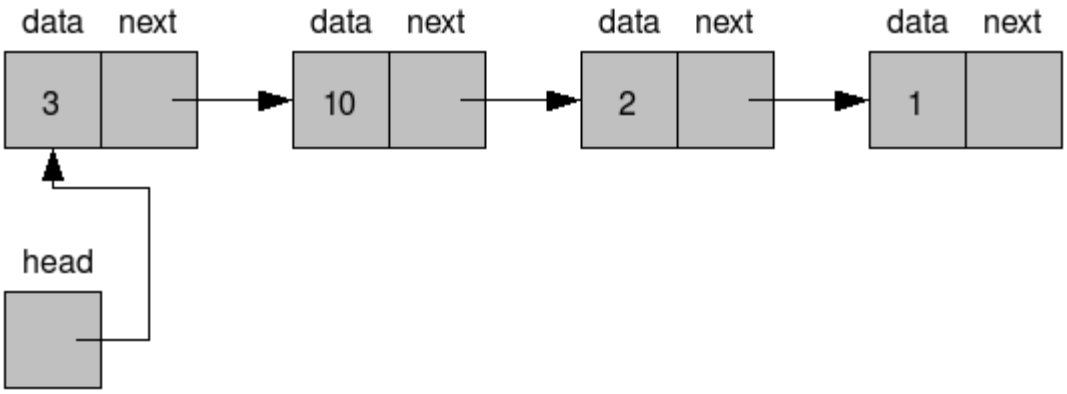
(B) Complete the code of Bank class as per the commented instructions

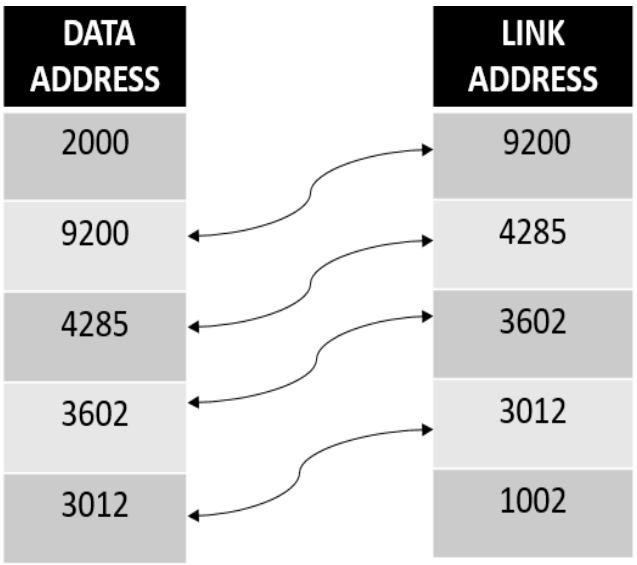
```
public class Bank {
private ArrayList<Account> accts;
int maxActive;

  public boolean addAccount (Account newone) {
/* Write the code for adding new account, return false if
account can't be created */
  }
  public boolean removeAccount (long acctnum) {
 /* Write the code for removing the account, return false if
account does not exist */
  }
  public double deposit(long acctnum, double amount) {
       /* Write the code for depositing specified amount to the
account,
       return -1 if account does not exist */
}
  public double withdraw(long acctnum, double amount) {
       /* Write the code for withdrawing specified amount from
the account,
       return -1 if insufficient balance or account does not
exist */
  }
 //override toString() method to display details of all the
accounts in bank
}
```

(C) Write a suitable driver class to test the behavior of the methods of above classes.

6. LinkedList

A linked list is a linear data structure where each element is a separate object. In a singly linked list each node in the list stores the contents of the node and a pointer or reference to the next node in the list. It does not store any pointer or reference to the previous node. It is called a singly linked list because each node only has a single link to another node. To store a single linked list, you only need to store a reference or pointer to the first node in that list. The last node has a pointer to nothingness to indicate that it is the last node.





It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the head is a null reference.

A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list.

Disadvantages of a Linked List:

- Does not allow direct access to the individual elements.
- Uses more memory compare with an array.

6.1 Node class

Node class is implemented as an inner class inside the Linked List class since it is to be purely used as a helper class

6.2 Operations on a LinkedList

Operation	Time Complexity
Insert at beginning	O(1)
Insert at end	O(N)
Delete at beginning	O(1)
Delete at end	O(N)
Search for an element	O(N)

Where N is the size of the linked list.

Linked List implementation:

```
import java.util.Iterator;
import java.util.ListIterator;

public class MyLinkedList {
    private Node head;     //denotes the head of the linked list
    private int size;      //denotes the size of the linked list

    // Constructors for Node Class
     public class Node{
        Object data;       //Data to be held by the node, could be integer or string
        Node next;         //Address of the next node in the linked list
     }
   public Node(Object data)
this.data = data;
       this.next=null;
    }

     public Node(Object data, Node next) {
this.data = data;
this.next = next;
}
     public String toString() {
String stringView = "";
stringView += '[';
stringView += data.toString();
stringView += ']';

           if (next != null) {
                stringView += '-';
           }
           return stringView;
        }
    }
```

```
    // Constructors for MyLinkedList Class
    public MyLinkedList() {
        size = 0;
    }
    public MyLinkedList(Node head) {
        this.head = head;
        size = 1;
    }
    public MyLinkedList(Object headData) {
        head = new Node(headData);
        size = 1;
    }

   // Methods

   //Getter and Setter methods for Head node
   public Node getHead() {
             return head;
   }
   public void setHead(Node head) {
             this.head = head;
   }
   public int getSize() {
             return size;
   }

//Adding new Node at the beginning of the list
 public void add(Object data) {
        Node e = new Node(data);
        e.next = head;
        head = e;
        size++;
    }
//Adding new Node at the end of the list
    public void addLast(Object data) {
        if (head == null)
            addFirst(data);
        else {
            Node x = head;
            while (x.next != null)
                x = x.next;
            x.next = new Node(data);
            size++;
        }
    }
    //Adding new Node at a particular index (Overloaded method)
    public void add(int index, Object data) {
        if(checkPositionIndex(index)==-1)return;

        if (index == 0)
            addFirst(data);
        else {
            Node x = head;
            for (int i = 0; i < index - 1; i++)
```

```
                x = x.next;
            x.next = new Node(data, x.next);
            size++;
        }
    }
     //Fetching a Node of a particular index
    public Object getElement(int index) {
        if( checkElementIndex(index)==-1)return null;
        Node x = head;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x.data;
    }
    //Removing the head Node of Linked List
    public Object removeFirst() {
        if (size == 0)
            return null;
        else {
            Object temp = head.data;
            head = head.next;
            size--;
            return temp;
        }
    }

//Removing the last Node of Linked List
    public Object removeLast() {
        if (size <= 1)
            return removeFirst();
        else {
            Node x = head;
            while (x.next.next != null)
                x = x.next;
            Object temp = x.next.data;
            x.next = null;
            size--;
            return temp;
        }
    }

   //Removing the a particular Node of LinkedList
    public void remove(int index) {
        if(checkElementIndex(index)==-1)return;
        if (index == 0)
            removeFirst();
        else {
            Node x = head;
            for (int i = 0; i < index - 1; i++)
                x = x.next;
            x.next = x.next.next;
            size--;
        }
    }
   //Changing data value of a particular Node
    public Object setElement(int index, Object newData) {
```

```
        if(checkElementIndex(index)==-1)return null;
        Node x = head;
        for (int i = 0; i < index; i++)
            x = x.next;
        Object temp = x.data;
        x.data = newData;
        return temp;
    }
//Returns the index of the first occurrence of the data
       public int indexOf(Object data) {
        Node x = head;
        for (int i = 0; i < size; i++, x = x.next)
            if (x.data.equals(data)) {
                return i;
            }
        return -1;
    }
//Returns the index of the last occurrence of the data
    public int lastIndexOf(Object data) {
        int lastOccurrence = -1;
        Node x = head;
        for (int i = 0; i < size; i++, x = x.next)
            if (x.data.equals(data))
                lastOccurrence = i;
        return lastOccurrence;
    }

//Helper Methods
   //Checks whether a particular object is there in the list
    public boolean contains(Object o) {
        for (Node x = head; x != null; x = x.next) {
            if (x.data.equals(o)) {
                return true;
            }
        }
        return false;
    }
  //Checks whether a particular index is feasible or not
    private int checkElementIndex(int index) {
        if (index < 0 || size <= index)
            return -1;
        return 1;
    }
   //Checks whether list is empty or not
    public boolean isEmpty() {
             return head == null;
    }
   // Returns the size of list
  public int getSize() {
        return size;
    }

    public String toString() {
        if (head == null) {
            return "null";
```

```
        } else {
            StringBuilder stringView = new StringBuilder();            // using
StringBuilder inside loops is very efficient
            for (Node x = head; x != null; x = x.next)
                stringView.append(x.toString());
            return stringView.toString();
        }
    }

    public String debugString() {
        return toString() + " (size: " + size + ")";
    }
}
```

Exercises:

1. Add a function in the MyLinkedList class to reverse the given linked list. The function should return the new head of the linked list.
2. Add a function in the MyLinkedList class to rotate the given linked list counter-clockwise by k nodes. The function should return the new head of the linked list. For example, if the given linked list is 10->20->30->40->50->60 and k is 4, the list should be modified to 50->60->10->20->30->40. Assume that k is smaller than the count of nodes in linked list and positive.

6.3 Iterator for a LinkedList

An Iterator can be made for the Linked List as an inner class for traversing the elements as necessary.

Implementing Iterator:

```
class SinglyLinkedListIterator implements Iterator{
       private Node current, lastAccessed;
       int index ;

       //Constructor for the Iterator inner class
       public SinglyLinkedListIterator() {
       current = head;
              lastAccessed = null;
              index = 0;
       }
       //Implementing methods for Iterator interface

       //Checks the feasibility of next position
       public boolean hasNext()      { return index < size; }
       // Returns the next index of iterator
       public int nextIndex()        { return index;     }
       //if possible, moves the iterator to the next element
       public Node next(){
              if (!hasNext()) return null;
              lastAccessed = current;
```

```
            Node node = current;
            current = current.next;
            index++;
            return node;
        }
        //Add new Node through iterator
        public void add(Object e) {
               Node x = lastAccessed;
               Node y = new Node(e,current);
               x.next = y;
               size++;
               index++;
               lastAccessed = null;
        }
        @Override
        //remove Node through iterator
        public void remove() {
               if (lastAccessed == null){
                      System.out.println("No Item to delete");
                      return;
               }
            Node x = lastAccessed;
            Node y = current.next;
            x.next = y;
            size--;
            if (current == lastAccessed)
                 current = y;
            else
                 index--;
            lastAccessed = null;
        }
        //Change the value of last Accessed Node
        public void set(Object e) {
        if (lastAccessed == null){
                      System.out.println("No Item to set");
                      return;
               }
            lastAccessed.data = e;
        }

    }
```

7. Doubly Linked List:

A **doubly linked list** is a list that has two references, one to the next node and another to previous node.Each node contains two fields, called links, that are references to the previous and to the next node in the sequence of nodes. The beginning and ending nodes previous and next links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list.

The two node links allow traversal of the list in either direction. While adding or removing a node in a doubly-linked list requires changing more links than the same operations on a singly linked list, the operations are simpler and potentially more efficient, because there is no need to keep track of the previous node during traversal or no need to traverse the list to find the previous node, so that its link can be modified.

Exercises:

1. Implement a Doubly Linked List in Java having all the functions implemented as shown for Singly Linked List.
2. Create an iterator for the same as an inner class.