Practical Exercise 6 – Lists, Stack and Queue – Solution

Question 1:

```java
import java.util.*;

public class TestMyList {
  public static void main(String[] args) {
    new TestMyList();
  }

  public TestMyList() {
    String[] name1 = {"Tom", "George", "Peter", "Jean", "Jane"};
    String[] name2 = {"Tom", "George", "Michael", "Michelle", "Daniel"};

    MyList<String> list1 = new MyArrayList<String>(name1);
    MyList<String> list2 = new MyArrayList<String>(name2);
    System.out.println("list1:" + list1);
    System.out.println("list2:" + list2);
    list1.addAll(list2);
    System.out.println("After addAll:");
    System.out.println("list1:" + list1);
    System.out.println("list2:" + list2);

    list1.retainAll(list2);
    System.out.println("After retainAll:");
    System.out.println("list1:" + list1);
    System.out.println("list2:" + list2);

    list1.removeAll(list2);
    System.out.println("After removeAll:");
    System.out.println("list1:" + list1);
    System.out.println("list2:" + list2);
  }

  public interface MyList<E> extends java.lang.Iterable {
    /** Add a new element at the end of this list */
    public void add(E e);

    /** Add a new element at the specified index in this list */
    public void add(int index, E e);

    /** Clear the list */
    public void clear();

    /** Return true if this list contains the element */
    public boolean contains(E e);

    /** Return the element from this list at the specified index */
    public E get(int index);

    /** Return the index of the first matching element in this list.
     *  Return -1 if no match. */
  public int indexOf(E e);

  /** Return true if this list contains no elements */
  public boolean isEmpty();

  /** Return the index of the last matching element in this list
   *  Return -1 if no match. */
  public int lastIndexOf(E e);

  /** Remove the first occurrence of the element o from this list.
   *  Shift any subsequent elements to the left.
   *  Return true if the element is removed. */
  public boolean remove(E e);

  /** Remove the element at the specified position in this list
   *  Shift any subsequent elements to the left.
   *  Return the element that was removed from the list. */
  public E remove(int index);

  /** Replace the element at the specified position in this list
   *  with the specified element and return the old element. */
  public Object set(int index, E e);

  /** Return the number of elements in this list */
  public int size();

     /** Adds the elements in otherList to this list
    * Returns true if this list changed as a result of the call */
  public boolean addAll(MyList<E> otherList);

  /** Removes all the elements in otherList from this list
    * Returns true if this list changed as a result of the call */
  public boolean removeAll(MyList<E> otherList);

  /** Retains the elements in this list that are also in otherList
    * Returns true if this list changed as a result of the call */
  public boolean retainAll(MyList<E> otherList);
}

public abstract class MyAbstractList<E> implements MyList<E> {
  protected int size = 0; // The size of the list

  /** Create a default list */
  protected MyAbstractList() {
  }

  /** Create a list from an array of objects */
  protected MyAbstractList(E[] objects) {
    for (int i = 0; i < objects.length; i++)
      add(objects[i]);
  }

  /** Add a new element at the end of this list */
  @Override
  public void add(E e) {
```

```
  add(size, e);
}

/** Return true if this list contains no elements */
@Override
public boolean isEmpty() {
  return size == 0;
}

/** Return the number of elements in this list */
@Override
public int size() {
  return size;
}

/** Remove the first occurrence of the element o from this list.
 *  Shift any subsequent elements to the left.
 *  Return true if the element is removed. */
@Override
public boolean remove(E e) {
  if (indexOf(e) >= 0) {
    remove(indexOf(e));
    return true;
  }
  else
    return false;
}

/** Adds the elements in otherList to this list.
 * Returns true if this list changed as a result of the call */
@Override
public boolean addAll(MyList<E> otherList) {
  for (int i = 0; i < otherList.size(); i++)
    add(otherList.get(i));

  if (otherList.size() > 0)
    return true;
  else
    return false;
}

/** Removes all the elements in otherList from this list
  * Returns true if this list changed as a result of the call */
@Override
public boolean removeAll(MyList<E> otherList) {
  boolean changed = false;
  for (int i = 0; i < otherList.size(); i++) {
    if (remove(otherList.get(i)))
      changed = true;
  }

  return changed;
}
```

```java
/** Retains the elements in this list that are also in otherList
    * Returns true if this list changed as a result of the call */
  @Override
  public boolean retainAll(MyList<E> otherList) {
    boolean changed = false;
    for (int i = 0; i < this.size(); ) {
      if (!otherList.contains(this.get(i))) {
        this.remove(get(i));
        changed = true;
      }
      else
        i++;
    }

    return changed;
  }
}

public class MyArrayList<E> extends MyAbstractList<E> {
  public static final int INITIAL_CAPACITY = 16;
  private E[] data = (E[])new Object[INITIAL_CAPACITY];


  /** Create a default list */
  public MyArrayList() {
  }

  /** Create a list from an array of objects */
  public MyArrayList(E[] objects) {
    for (int i = 0; i < objects.length; i++)
      add(objects[i]); // Warning: don’t use super(objects)!
  }

  /** Add a new element at the specified index in this list */
  @Override
  public void add(int index, E e) {
    ensureCapacity();

    // Move the elements to the right after the specified index
    for (int i = size - 1; i >= index; i--)
      data[i + 1] = data[i];

    // Insert new element to data[index]
    data[index] = e;

    // Increase size by 1
    size++;
  }

  /** Create a new larger array, double the current size + 1 */
  private void ensureCapacity() {
    if (size >= data.length) {
      E[] newData = (E[])(new Object[size * 2 + 1]);
      System.arraycopy(data, 0, newData, 0, size);
      data = newData;
  }
}

/** Clear the list */
@Override
public void clear() {
  data = (E[])new Object[INITIAL_CAPACITY];
  size = 0;
}

/** Return true if this list contains the element */
@Override
public boolean contains(Object e) {
  for (int i = 0; i < size; i++)
    if (e.equals(data[i])) return true;

  return false;
}

/** Return the element from this list at the specified index */
@Override
public E get(int index) {
  return data[index];
}

/** Return the index of the first matching element in this list.
 *  Return -1 if no match. */
@Override
public int indexOf(Object e) {
  for (int i = 0; i < size; i++)
    if (e.equals(data[i])) return i;

  return -1;
}

/** Return the index of the last matching element in this list
 *  Return -1 if no match. */
@Override
public int lastIndexOf(Object e) {
  for (int i = size - 1; i >= 0; i--)
    if (e.equals(data[i])) return i;

  return -1;
}

/** Remove the element at the specified position in this list
 *  Shift any subsequent elements to the left.
 *  Return the element that was removed from the list. */
@Override
public E remove(int index) {
  E e = data[index];

  // Shift data to the left
  for (int j = index; j < size - 1; j++)
    data[j] = data[j + 1];
```

```java
  data[size - 1] = null; // This element is now null

  // Decrement size
  size--;

  return e;
}

/** Replace the element at the specified position in this list
 *  with the specified element. */
@Override
public E set(int index, E e) {
  E old = data[index];
  data[index] = e;
  return old;
}

@Override
public String toString() {
  StringBuilder result = new StringBuilder("[");

  for (int i = 0; i < size; i++) {
    result.append(data[i]);
    if (i < size - 1) result.append(", ");
  }

  return result.toString() + "]";
}

/** Trims the capacity to current size */
public void trimToSize() {
  if (size != data.length) {
    E[] newData = (E[])(new Object[size]);
    System.arraycopy(data, 0, newData, 0, size);
    data = newData;
  } // If size == capacity, no need to trim
}

/** Override the iterator method defined in Iterable */
@Override
public java.util.Iterator<E> iterator() {
  return new ArrayListIterator();
}

private class ArrayListIterator
    implements java.util.Iterator<E> {
  private int current = 0; // Current index

  @Override
  public boolean hasNext() {
    return (current < size);
  }

  @Override
```

```
      public E next() {
        return data[current++];
      }

      @Override
      public void remove() {
        MyArrayList.this.remove(current);
      }
    }

    @Override
    public int size() {
      // TODO Auto-generated method stub
      return size;
    }
  }
}
```

Question 2:

```
import java.util.*;
import java.util.ArrayList;

public class TestMyLinkedList {
    public static void main(String[] args) {
        new TestMyLinkedList();
    }

    public TestMyLinkedList() {
        String[] name = {"Tom", "George", "Peter", "Jean", "George", "Jane"};
        MyList<String> list = new MyLinkedList<String>(name);

        System.out.println(list.contains("George"));
        System.out.println(list.get(3));
        System.out.println(list.indexOf("George"));
        System.out.println(list.lastIndexOf("George"));
        list.set(4, "Michael");
        System.out.println(list);
    }

    public interface MyList<E> extends java.lang.Iterable {
        /** Add a new element at the end of this list */
        public void add(E e);

        /** Add a new element at the specified index in this list */
        public void add(int index, E e);

        /** Clear the list */
        public void clear();

        /** Return true if this list contains the element */
        public boolean contains(E e);

        /** Return the element from this list at the specified index */
        public E get(int index);

        /** Return the index of the first matching element in this list.
         *  Return -1 if no match. */
        public int indexOf(E e);

        /** Return true if this list contains no elements */
        public boolean isEmpty();

        /** Return the index of the last matching element in this list
         *  Return -1 if no match. */
        public int lastIndexOf(E e);

        /** Remove the first occurrence of the element o from this list.
         *  Shift any subsequent elements to the left.
         *  Return true if the element is removed. */
        public boolean remove(E e);
    /** Remove the element at the specified position in this list
     *  Shift any subsequent elements to the left.
     *  Return the element that was removed from the list. */
    public E remove(int index);

    /** Replace the element at the specified position in this list
     *  with the specified element and return the old element. */
    public Object set(int index, E e);

    /** Return the number of elements in this list */
    public int size();

     /** Return an iterator for the list */
    public java.util.Iterator<E> iterator();
}

public abstract class MyAbstractList<E> implements MyList<E> {
    protected int size = 0; // The size of the list

    /** Create a default list */
    protected MyAbstractList() {
    }

    /** Create a list from an array of objects */
    protected MyAbstractList(E[] objects) {
        for (int i = 0; i < objects.length; i++)
            add(objects[i]);
    }

    /** Add a new element at the end of this list */
    @Override
    public void add(E e) {
        add(size, e);
    }

    /** Return true if this list contains no elements */
    @Override
    public boolean isEmpty() {
        return size == 0;
    }

    /** Return the number of elements in this list */
    @Override
    public int size() {
        return size;
    }

    /** Remove the first occurrence of the element o from this list.
     *  Shift any subsequent elements to the left.
     *  Return true if the element is removed. */
    @Override
    public boolean remove(E e) {
        if (indexOf(e) >= 0) {
            remove(indexOf(e));
            return true;
        }
        else
            return false;
    }
}

public class MyLinkedList<E> extends MyAbstractList<E> {
    private Node<E> head, tail;

    /** Create a default list */
    public MyLinkedList() {
    }

    /** Create a list from an array of objects */
    public MyLinkedList(E[] objects) {
        super(objects);
    }

    /** Return the head element in the list */
    public E getFirst() {
        if (size == 0) {
            return null;
        }
        else {
            return head.element;
        }
    }

    /** Return the last element in the list */
    public E getLast() {
        if (size == 0) {
            return null;
        }
        else {
            return tail.element;
        }
    }

    /** Add an element to the beginning of the list */
    public void addFirst(E e) {
        Node<E> newNode = new Node<E>(e); // Create a new node
        newNode.next = head; // link the new node with the head
        head = newNode; // head points to the new node
        size++; // Increase list size

        if (tail == null) // the new node is the only node in list
            tail = head;
        }

        /** Add an element to the end of the list */
        public void addLast(E e) {
            Node<E> newNode = new Node<E>(e); // Create a new for element e

            if (tail == null) {
                head = tail = newNode; // The new node is the only node in list
            }
            else {
                tail.next = newNode; // Link the new with the last node
                tail = tail.next; // tail now points to the last node
            }

            size++; // Increase size
        }


        /** Add a new element at the specified index in this list
         *  The index of the head element is 0 */
        @Override
        public void add(int index, E e) {
            if (index == 0) {
            addFirst(e);
           }
            else if (index >= size) {
                addLast(e);
            }
            else {
                Node<E> current = head;
                for (int i = 1; i < index; i++) {
                current = current.next;
            }
            Node<E> temp = current.next;
            current.next = new Node<E>(e);
            (current.next).next = temp;
            size++;
        }
    }

    /** Remove the head node and
     *  return the object that is contained in the removed node. */
    public E removeFirst() {
        if (size == 0) {
            return null;
        }
        else {
            Node<E> temp = head;
            head = head.next;
            size--;
            if (head == null) {
                tail = null;
            }
            return temp.element;
        }
    }

    /** Remove the last node and
     *  return the object that is contained in the removed node. */
    public E removeLast() {
        if (size == 0) {
            return null;
        }
        else if (size == 1) {
            Node<E> temp = head;
            head = tail = null;
            size = 0;
            return temp.element;
        }
        else {
            Node<E> current = head;

            for (int i = 0; i < size - 2; i++) {
                current = current.next;
            }

            Node<E> temp = tail;
            tail = current;
            tail.next = null;
            size--;
            return temp.element;
        }
    }

    /** Remove the element at the specified position in this list.
     *  Return the element that was removed from the list. */
    @Override
    public E remove(int index) {
        if (index < 0 || index >= size) {
            return null;
        }
        else if (index == 0) {
            return removeFirst();
        }
        else if (index == size - 1) {
            return removeLast();
        }
        else {
            Node<E> previous = head;

            for (int i = 1; i < index; i++) {
                previous = previous.next;
            }

            Node<E> current = previous.next;
            previous.next = current.next;
            size--;
            return current.element;
        }
    }

    /** Clear the list */
    @Override
    public void clear() {
        head = tail = null;
    }

    /** Return true if this list contains the element o */
```

```
@Override
public boolean contains(E e) {
// Implement it in this exercise
    Node<E> current = head;
    for (int i = 0; i < size; i++) {
        if (current.element.equals(e))
            return true;
        current = current.next;
    }

    return false;
}

/** Return the element from this list at the specified index */
@Override
public E get(int index) {
    // Implement it in this exercise
    if (index < 0 || index > size - 1)
        return null;

    Node<E> current = head;
    for (int i = 0; i < index; i++)
        current = current.next;

    return current.element;
}

/** Returns the index of the first matching element in this list.
 *  Returns -1 if no match. */
@Override
public int indexOf(E e) {
    // Implement it in this exercise
    Node<E> current = head;
    for (int i = 0; i < size; i++) {
        if (current.element.equals(e))
            return i;
        current = current.next;
    }

    return -1;
}

/** Returns the index of the last matching element in this list
 *  Returns -1 if no match. */
@Override
public int lastIndexOf(E e) {
// Implement it in this exercise
    int lastIndex = -1;
    Node<E> current = head;
    for (int i = 0; i < size; i++) {
        if (current.element.equals(e))
            lastIndex = i;
        current = current.next;
    }
```

```
        return lastIndex;
    }

    /** Replace the element at the specified position in this list
     *  with the specified element. */
    @Override
    public E set(int index, E e) {
        if (index < 0 || index > size - 1)
            return null;

        Node<E> current = head;
        for (int i = 0; i < index; i++)
            current = current.next;

        E temp =  current.element;
        current.element = e;

        return temp;
    }

    @Override
    public String toString() {
        StringBuilder result = new StringBuilder("[");

        Node<E> current = head;
        for (int i = 0; i < size; i++) {
            result.append(current.element);
            current = current.next;
            if (current != null) {
                result.append(", "); // Separate two elements with a comma
            }
            else {
                result.append("]"); // Insert the closing ] in the string
            }
        }

        return result.toString();
    }

    /** Override the iterator method defined in Iterable */
    @Override
    public java.util.Iterator<E> iterator() {
        return new LinkedListIterator();
    }

    private class LinkedListIterator implements java.util.Iterator<E> {
        private Node<E> current = head; // Current index

        @Override
        public boolean hasNext() {
            return (current != null);
        }

        @Override
        public E next() {
                E e = current.element;
                current = current.next;
                return e;
            }

            @Override
            public void remove() {
                System.out.println("Implementation left as an exercise");
            }
        }

        private class Node<E> {
            E element;
            Node<E> next;

            public Node(E element) {
                this.element = element;
            }
        }
    }
}
```

Question 3:

```
import java.util.Scanner;
import java.util.Stack;

public class ReverseTextWithStackQueue {
     public static void main(String[] args) {
         new ReverseTextWithStackQueue();
     }

     public ReverseTextWithStackQueue() {
         Stack<Character> stack = new Stack<Character>();
         GenericQueue<Character> queue = new GenericQueue<Character>();

         Scanner in = new Scanner(System.in);
         String text = in.nextLine();

         for(int i = 0; i < text.length(); i++) {
             queue.enqueue(text.charAt(i));
         }

         while(queue.getSize() != 0) {
             stack.push(queue.dequeue());
         }

         while(!stack.empty()) {
             System.out.print(stack.pop() + " ");
         }
         System.out.print("\n");
     }

     public class GenericQueue<E> {
        private java.util.LinkedList<E> list
            = new java.util.LinkedList<E>();

        public void enqueue(E e) {
            list.addLast(e);
        }

        public E dequeue() {
            return list.removeFirst();
        }

        public int getSize() {
            return list.size();
        }

        @Override
        public String toString() {
            return "Queue: " + list.toString();
        }
    }
}
```

Question 4:

```
import java.util.Scanner;

public class EvaluatePrefix {
    public static void main(String[] args) {
        new EvaluatePrefix();
    }

    public EvaluatePrefix() {
        GenericQueue<String> prefixQ = new GenericQueue<String>();

        Scanner in = new Scanner(System.in);
        String text = in.nextLine();

        for(int i = 0; i < text.length(); i++) {
            prefixQ.enqueue(text.charAt(i) + "");
        }

        while(prefixQ.getSize() > 1) {
            GenericQueue<String> tempQ = new GenericQueue<String>();
            while(prefixQ.getSize() > 0) {
                String item = prefixQ.dequeue();
                if(Character.isDigit(item.charAt(0))) {
                    tempQ.enqueue(item);
                }
                else {
                    char operator = item.charAt(0);
                    if(!Character.isDigit(prefixQ.getFront().charAt(0))) {
                        tempQ.enqueue(operator + "");
                    }
                    else {
                        double operand1 = Double.parseDouble(prefixQ.dequeue());
                        if(!Character.isDigit(prefixQ.getFront().charAt(0))){
                            tempQ.enqueue(operator + "");
                            tempQ.enqueue(operand1 + "");
                        }
                        else {
                            double operand2 = Double.parseDouble(prefixQ.dequeue());
                            double value;
                            if(operator == '+')
                                value = operand1 + operand2;
                            else if(operator == '-')
                                value = operand1 - operand2;
                            else if(operator == '*')
                                value = operand1 * operand2;
                            else
                                value = operand1 / operand2;
                            tempQ.enqueue(value + "");
                        }
                    }
                }
            }
            prefixQ = tempQ;
            System.out.println(prefixQ.toString());
        }
        System.out.println("The answer is " + prefixQ.dequeue());
    }

    public class GenericQueue<E> {
        private java.util.LinkedList<E> list
            = new java.util.LinkedList<E>();

        public void enqueue(E e) {
            list.addLast(e);
        }

        public E dequeue() {
            return list.removeFirst();
        }

        public E getFront() {
            return list.getFirst();
        }

        public E getRear() {
            return list.getLast();
        }

        public int getSize() {
            return list.size();
        }

        @Override
        public String toString() {
            return "Queue: " + list.toString();
        }
    }
}
```