# LAB 2 - Stack & Queue

Junjie Zhang

25 Sep 2019

## Introduction

In this lab , your task is to implement stack and queue class. The task can be divided into two parts. in Part 1, you need to implement the **double-ended queue** with doubly linked list. In Part 2, you need to implement stack and queue class with **double-ended queue of Part 1**.

**Note that**, you should implement your own data structure and may not call any library function other than those in `java.lang` in this lab.

### Part 1

To model a single node in a doubly linked list, you can create a data type `DLNode` with the following API:

```
public class DLNode {
    // create a DLNode
    DLNode(int element)
    // get the value of the element
    public int getElement()
    // set the value of the element
    public void setElement(int element)
    // get the previous DLNode
    public DLNode getPrev()
    // set the previous DLNode
    public void setPrev(DLNode prev)
    // get the next DLNode
    public DLNode getNext()
    // set the next DLNode
    public void setNext(DLNode next)
}
```

Then, you can use `DLNode` to create a double-ended queue with the following API:

```
public class MyDeque {
    //create a deque
    MyDeque(DLNode node)
    //insert a node at the beginning of deque
    public void insertFirst(DLNode node)
    //remove and return the first node
    public DLNode removeFirst()
    //insert a node at the end of deque
    public void insertLast(DLNode node)
    //remove and return the last node
    public DLNode removeLast()
    //return first node
    public DLNode first()
    //return last node
    public DLNode last()
```

```
    //return number of nodes
    public int size()
    //judge whether the deque is empty
    public boolean isEmpty()
    //display content of the deque
    public String toString()
}
```

### Part 2

Now, you have implemented the double-ended queue. In this part, you need to implement the data type `MyStack` and `MyQueue` with `MyDeque` by using adaptor pattern. Your task is to design the `StackAdaptor` and `QueueAdaptor` which inherit `MyStack` and `MyQueue` respectively.

Namely, the data type `MyStack` and `MyQueue` are abstract, you can create a class called `StackAdaptor` to extend the abstract class `MyStack` using your `MyDeque`, and so is `QueueAdaptor`.

```
public abstract class MyStack {
    //push a node into stack
    public abstract void push(DLNode node);
    //pop a node from stack
    public abstract DLNode pop();
    //return top node
    public abstract DLNode top();
    //return number of nodes
    public abstract int size();
    //judge whether the stack is empty
    public abstract boolean isEmpty();
    //display the content of the stack
    public abstract String toString();
}
```

```
public abstract class MyQueue {
    //enqueue a node
    public abstract void enqueue(DLNode node);
    //dequeue a node
    public abstract DLNode dequeue();
    //return front node of queue
    public abstract DLNode front();
    //return number of nodes
    public abstract int size();
    //judge whether the queue is empty
    public abstract boolean isEmpty();
    //display the content of the stack
    public abstract String toString();
}
```

## Test case

The following are examples to test the data types you implement. After each operation, there is an annotation showing the state of the double-ended queue or the stack or the queue. Test1 is for part 1 and Test 2 is for part 2.

```
public class Test1 {
    public static void main(String[] args) {
        DLNode tmp = null;
        System.out.println("Deque");
        MyDeque myDeque = new MyDeque(new DLNode(1));
        System.out.println("1: " + myDeque.toString()); // 1: 1
        myDeque.insertFirst(new DLNode(2));
        System.out.println("2: " + myDeque.toString()); // 2: 2 1
        myDeque.insertLast(new DLNode(3));
        System.out.println("3: " + myDeque.toString()); // 3: 2 1 3
        tmp = myDeque.removeFirst();
        System.out.println("remove first " + tmp.getElement()); // remove first
2
        System.out.println("4: " + myDeque.toString()); // 4: 1 3
        tmp = myDeque.removeLast();
        System.out.println("remove last " + tmp.getElement()); // remove last 3

        System.out.println("5: " + myDeque.toString()); // 5: 1
    }
}
```

```
public class Test2 {
    public static void main(String[] args) {
        DLNode tmp = null;
        System.out.println("Stack");
        MyStack stack = new StackAdaptor(new MyDeque(new DLNode(1)));
        System.out.println("1: " + stack.toString()); // 1: 1
        stack.push(new DLNode(2));
        System.out.println("2: " + stack.toString()); // 2: 1 2
        tmp = stack.pop();
        System.out.println("pop " + tmp.getElement()); // pop 2
        System.out.println("3: " + stack.toString()); // 3: 1
        tmp = stack.pop();
        System.out.println("pop " + tmp.getElement()); // pop 1
        System.out.println("4: " + stack.toString()); // 4:
        stack.push(new DLNode(3));
        System.out.println("5: " + stack.toString()); // 5: 3
        System.out.println("Queue");
        MyQueue queue = new QueueAdaptor(new MyDeque(new DLNode(1)));
        System.out.println("1: " + queue.toString()); // 1 : 1
        tmp = queue.dequeue();
        System.out.println("dequeue " + tmp.getElement()); // dequeue 1
        System.out.println("2: " + queue.toString()); // 2:
        queue.enqueue(new DLNode(2));
        System.out.println("3: " + queue.toString()); // 3: 2
        queue.enqueue(new DLNode(3));
        System.out.println("4: " + queue.toString()); // 4: 2 3
        tmp = queue.dequeue();
        System.out.println("dequeue " + tmp.getElement()); //dequeue 2
        System.out.println("5: " + queue.toString()); // 5: 3
    }
}
```

## Bonus : Card game

Now,  we want to design a card game called "solitaire" or "jie long" in Chinese. You can implement it with `StackAdaptor` before. The rule is when a new node is inserted into the stack, if there is one node in the stack that has the same element with the new node, then nodes between these two nodes (contain the two nodes) will be cleared from the stack. You can add new functions to `StackAdaptor` or just do it in main function. The following is a demo:

`Stack: 1 2 3 4 5`

`Insert 4`

`Stack: 1 2 3`

`Insert 7`

`Stack: 1 2 3 7`

`Insert 2`

`Stack: 1`

## Deadline

> 2019/9/27 18:00
>
> Please compress your code and upload into the FTP server with filename in format YourStudentID.zip.