Lab Tasks 1 – Solution

1. Exercise 3.16, page 97: Download the assignment project from `http://www.comp.nus.edu.sg/~cs1102s/java/labtasks_01.zip`.

   Complete the class reverseIterator .MyArrayListWithReverse.java.

   The given program MyArrayList.java is taken from the textbook.

   The iterator function of MyArrayListWithReverse should handle the following exceptions:

   - next() throws the exception java. util .NoSuchElementException,
   - remove() throws the exception IllegalStateException if remove() is called without an immediately preceding next().

   (You may ignore exceptions arising from concurrent modification java. util .ConcurrentModificationException.)

   Solution: The implementation of the list ADT in MyArrayListWithReverseis the same as in MyArraList, the difference is only in the iterator:

   - The "normal" iterator initialized to start at the front element and traverses to the end of the list. The reverse iterator is initialized to the end of the list (array index size()-1) and traverses the list to the front.
   - The boolean flag okToRemove ensures that remove() can only be called after a preceding next() call.

```
private class ArrayListReverseIterator
                        implements java.util.Iterator<AnyType> {
   private int current = size()-1;
   boolean okToRemove = false;

   public boolean hasNext( ){
     return current >= 0;
   }

   public AnyType next( ) {
     if( !hasNext( ) )
       throw new java.util.NoSuchElementException( );
       okToRemove = true;
       return theItems[ current-- ];
     }

   public void remove( ) {
     if (! okToRemove) {
throw new java.lang.IllegalStateException();
     }
     okToRemove = false;
     MyArrayListWithReverse.this.remove( ++current );
```

```
    }
}
```

2. Exercise 3.21 (b), page 98: Implement class balancing.CheckBalanced.java. Note the following facts about Java comments:

   - When the compiler reads /∗, it skips any text until the next character sequence ∗/ (ignore all brackets between these two "tokens").
   - When the compiler reads //, it skips any text until the next newline character (ignore all brackets in between).

   You may assume that the given Java program has no strings.

   Solution:

   The first thing the program has to do is to parse the input into a sequence of tokens that can be processed to check whether all parentheses and comments are well balanced or not. There are multiple ways to implement the parsing step. In my solution, I use a simple tokenizer that reads ahead one character and checks whether it found any of the two symbol comment tokens (// , /* , */). If so, it return the two symbol token, otherwise it returns one character at a time.

   To keep track of whether we are inside a comment or not, we can simulate a finite state automata (FSA) with three states:

```
// FSA states to parse java comments
private final static int NOCOMMENT = 0;
private final static int SLASHSLASHCOMMENT = 1;
private final static int SLASHSTARCOMMENT = 2;
private static int state;
```

   The transitions of the FSA (the rules how to change from one state to another) are as follows:

```
//FSA rules to parse java comments
 if ((state == NOCOMMENT) && ("//".equals(token)))
        state = SLASHSLASHCOMMENT;
 else if ((state == NOCOMMENT) && ("/*".equals(token))) {
                        state = SLASHSTARCOMMENT;
        myStack.push("/*");
 } else if ((state == SLASHSLASHCOMMENT) && (("\n".equals(token)) ||
                                                ("\r".equals(token))))
        state = NOCOMMENT;
 else if ((state == SLASHSTARCOMMENT) && ("*/".equals(token)))
        state = NOCOMMENT;
```

   The next thing is to check whether all expression in the input are balanced. This can be implemented using a stack ADT as you have seen during the lecture. Whenever the FSA sees an opening token (/*, (or [ ), it pushes the symbol on the stack. Whenever it sees a closing token (*/,),]),

it checks whether there is a matching opening item on top of the stack. If not, the Java program is surely not well balanced. If yes, remove the top item from the stack and continue. If the FSA reads the whole input successfully and the stack is empty at the end of the execution, the input is well balanced.

```
// check for balanced expressions
if ((state == NOCOMMENT) && ("{".equals(token) || "(".equals(token)
                                                || "[".equals(token))){
     // token is opening parentheses
     myStack.push(token);
} else if ((state == NOCOMMENT) && ("}".equals(token) ||")".equals(token)
                          || "]".equals(token) || "*/".equals(token))){
     // token is closing parentheses
     if (myStack.isEmpty()) {
            // unbalanced number of opening and closing parentheses
             return false;
     }
     boolean matches = checkPar(myStack.pop(), token);
     if (!matches) {
             return false;
     }
}
```

If the program reads the complete input without returning false early, then the expression is balanced if and only if the stack is empty, i.e. if there is a matching closing token for every opening token.

Note that there are many other syntax requirements for real Java code that our program does not check, e.g. curly parenthesis, string literals, etc.

3. Exercise 3.24, page 98: Implement the class twoStacks.MyTwoStacksArray.java.

   Solution:

   Because a stack can only grow in one direction, you can easily implement two stacks in one array by letting them start at the opposite ends of the array: one stack grows from the front of the array to the back, the other one from the back to the front. To keep track of the top element in each stack, you need two pointers.

```
private final int theSize = 10;
private Any[] myArray = (Any[]) new Object[10];
private int top1 = -1;
private int top2 = theSize;
```

   The stack operations push, pop, top, empty behave as before, only that you have each operation for each of the two stacks, e.g. pop1 and pop2.

```
public Any push1(Any item) {
    top1 += 1;
```

```
    if (top1 == top2) {
            throw new StackOverflowError();
    }
    myArray[top1] = item;
    return item;
}

public Any push2(Any item) {
    top2 -= 1;
    if (top2 == top1) {
            throw new StackOverflowError();
    }
    myArray[top2] = item;
    return item;
}
```

4. Implement a queue data structure as described in the textbook, using arrays, where the front and back pointers wrap around. When enqueue(..) is attempted on a queue whose array is full with queue elements, resize the array as with ArrayList.

   Implement the class queues.MyArrayQueue.java.

   Solution:

   The main difference between a queue and a stack is that a stack is a LIFO (last in first out) buffer and a queue is a FIFO (first in first out) buffer. Because a queue can be manipulated at both ends (enqueue at the back, dequeue at the front), you need to keep two indexes to the both ends of the queue.

   ```
   private int front;
   private int back;
   private Any [ ] theItems;
   ```

   The size of the queue can be calculated as the back index minus the front index. However be careful that the index can "wrap around" and the size suddenly becomes negative or larger than array length -1. In that case, we need to add the length of the array. This corresponds to a modulo operation % theItems.length. (Java's modulo operator does behave different for negative numbers, that is why it is not used here)

   ```
   public int size() {
       // the size of the queue is
       // (back index - front index + 1) % array.length
       int size = (back - front + 1);
       if (size < 0) {
               size += theItems.length;
       } else if (size >= theItems.length) {
               size -= theItems.length;
       }
   ```

```
    return size;
}
```

When the a new element is added to the queue, the operation first checks whether the underlying array is full and has to be enlarged. We will see later how this is done. Otherwise, the back index is increased by one modulo the array length and the new item is added at the back position.

```
public void enqueue(Any item) {
   if (size() == theItems.length -1) {
        // enlarge array size
        ensureCapacity( size( ) * 2 + 1 );
   }
   back = (back + 1) % theItems.length;
   theItems[back] = item;
}
```

When the an element is removed from the queue, the operation first checks whether the queue is empty. If so, it throws an exception. Otherwise, the front index is increased by one modulo the array length.

```
public Any dequeue() {
   if (empty()) {
        throw new NoSuchElementException();
   } else {
        Any theItem = theItems[front];
        front = (front + 1) % theItems.length;
        return theItem;
   }
}
```

The last thing we need to do is ensure that the arrays is always big enough to hold the queue. The idea is the same as for the MyArrayList in the book: whenever the arrays is full, create a new array of double the size and copy the elements from the old array over. In the case of the queue, we can do this by starting from the front index and copying the first size() elements over. Note the modulo old.length operation that wraps the index around if necessary.

```
public void ensureCapacity( int newCapacity ) {

   if( newCapacity < theItems.length )
        return;

   Any [ ] old = theItems;
   int oldSize = size();
   theItems = (Any []) new Object[ newCapacity ];
   //copy items into the new array
   for( int i = 0; i < oldSize; i++ )
        theItems[ i ] = old[ (front + i) % old.length];
```

```
        front = 0;
        back = oldSize-1;
        if (back < 0) {
            back += theItems.length;
        }
    }
```