Assignment 5:

Problem 1:

Write a program that takes as input an arithmetic expression. The program outputs whether the expression contains matching grouping symbols. For example, the arithmetic expressions {25 + (3 – 6) \* 8} and 7 + 8 \* 2 contains matching grouping symbols. However, the expression 5 + {(13 + 7) / 8 - 2 \* 9 does not contain matching grouping symbols.

package sameArithmeticPattern;

import java.util.ArrayList;

import java.util.List;

import java.util.stream.IntStream;

public class SameAritheticPattern

{

public static boolean getPattern(String expression1, String expression2)

{

String betweenBrackets = "";

String outsideOfBrackets = "";

String thePattern = "";

String thePattern1 = "";

if (expression1.contains("(") && expression1.contains(")"))

{

for(int i = expression1.indexOf('('); i <= expression1.indexOf(')'); i++)

{

betweenBrackets += expression1.charAt(i);

}

outsideOfBrackets = expression1.replace(betweenBrackets, "#");

outsideOfBrackets.trim();

for (int j = 0; j < outsideOfBrackets.length(); j++)

{

switch(outsideOfBrackets.charAt(j))

{

case '+':

thePattern += outsideOfBrackets.charAt(j);

break;

case '-':

thePattern += outsideOfBrackets.charAt(j);

break;

case '/':

thePattern += outsideOfBrackets.charAt(j);

break;

case '\*':

thePattern += outsideOfBrackets.charAt(j);

break;

}

}

//System.out.println(thePattern);

}

else

{

for (int j = 0; j < expression1.length(); j++)

{

switch(outsideOfBrackets.charAt(j))

{

case '+':

thePattern += outsideOfBrackets.charAt(j);

break;

case '-':

thePattern += outsideOfBrackets.charAt(j);

break;

case '/':

thePattern += outsideOfBrackets.charAt(j);

break;

case '\*':

thePattern += outsideOfBrackets.charAt(j);

break;

}

}

}

if (expression2.contains("(") && expression2.contains(")"))

{

for(int i = expression2.indexOf('('); i <= expression2.indexOf(')'); i++)

{

betweenBrackets += expression2.charAt(i);

}

outsideOfBrackets = expression2.replace(betweenBrackets, "#");

outsideOfBrackets.trim();

for (int j = 0; j < outsideOfBrackets.length(); j++)

{

switch(outsideOfBrackets.charAt(j))

{

case '+':

thePattern1 += outsideOfBrackets.charAt(j);

break;

case '-':

thePattern1 += outsideOfBrackets.charAt(j);

break;

case '/':

thePattern1 += outsideOfBrackets.charAt(j);

break;

case '\*':

thePattern1 += outsideOfBrackets.charAt(j);

break;

}

}

//System.out.println(thePattern);

}

else

{

for (int j = 0; j < expression2.length(); j++)

{

switch(outsideOfBrackets.charAt(j))

{

case '+':

thePattern1 += outsideOfBrackets.charAt(j);

break;

case '-':

thePattern1 += outsideOfBrackets.charAt(j);

break;

case '/':

thePattern1 += outsideOfBrackets.charAt(j);

break;

case '\*':

thePattern1 += outsideOfBrackets.charAt(j);

break;

}

}

}

if (thePattern.equals(thePattern1))

{

return true;

}

return false;

}

}

class Tester

{

public static void main(String[] args)

{

String expression1 = "25 + (3 – 6) \* 8";

String expression2 = "7 + 8 \* 2";

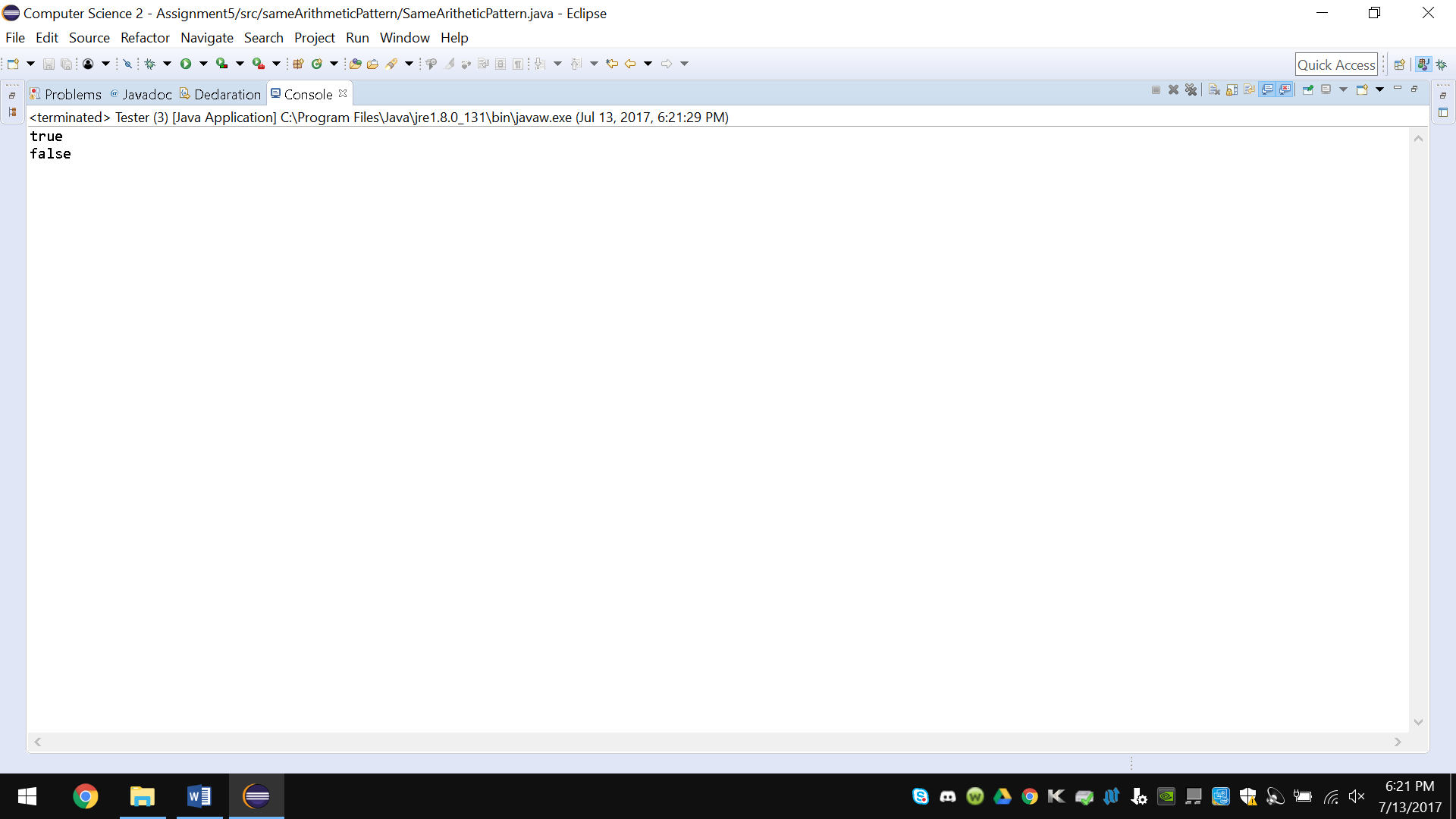
String expression3 = "5 + {(13 + 7) / 8 - 2 \* 9";

System.out.println(SameAritheticPattern.getPattern(expression1, expression2));

System.out.println(SameAritheticPattern.getPattern(expression1, expression3));

}

}



Problem 2:

Write a program that reads a line of text, changes each uppercase letter to lowercase, and places each letter both in a queue and onto a stack. The program should then verify whether the line of text is a palindrome (a set of letters or numbers that is the same whether read forward or backward).

**package** isPalindrome;

**import** java.util.\*;;

**public** **class** IsPalidrome

{

Queue<String> theQ = **new** LinkedList<>();

**public** IsPalidrome(Queue<String> theQ)

{

**this**.theQ = theQ;

**this**.toLowerCase();

System.***out***.println(**this**.listOfPalidrone());

}

**private** **void** toLowerCase()

{

Queue<String> lowerQ = **new** LinkedList<String>();

**for**(Iterator<String> it = theQ.iterator(); it.hasNext();)

{

lowerQ.add(it.next().toLowerCase());

}

theQ = lowerQ;

}

**private** List<Boolean> listOfPalidrone()

{

List<Boolean> list = **new** ArrayList<Boolean>();

**for**(Iterator<String> it = theQ.iterator(); it.hasNext();)

{

String next = it.next();

**if**(**new** StringBuilder().append(next.toLowerCase()).reverse().toString().equals(next.toLowerCase()))

{

list.add(**true**);

}

**else**

{

list.add(**false**);

}

}

**return** list;

}

}

**class** Tester

{

**public** **static** **void** main(String[] args)

{

Queue theQ = **new** LinkedList<String>();

theQ.add("aaaaa");

theQ.add("aaabb");

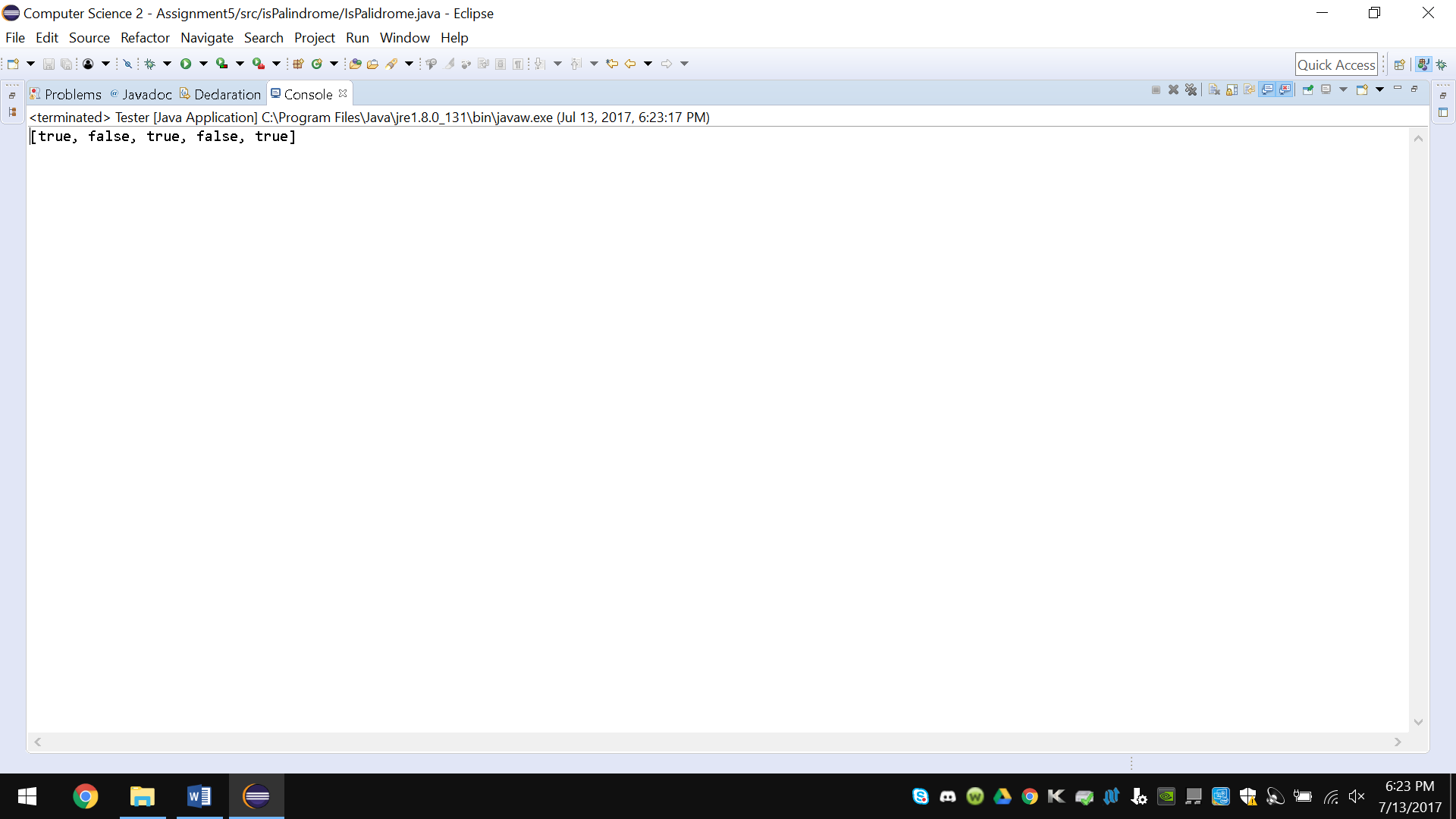
theQ.add("aabaa");

theQ.add("bbaaa");

theQ.add("bbabb");

IsPalidrome palidrome = **new** IsPalidrome(theQ);

}

} 

Problem 3:

Write a method areEqual that returns true if the two stacks specified in the parameters have the same elements in the same order, and false otherwise. Two elements are the same if they refer to the same object. The method may remove elements from the stacks, but it must return the elements to the stacks in the same order to restore the stacks to their original state. The only additional data structure that it can use as auxiliary storage is a single stack

package areEqual;

import java.util.Iterator;

import java.util.Stack;

public class AreEqualStacks

{

Stack<Object> stack = new Stack<>();

public static boolean areEqual(Stack s, Stack s1)

{

Iterator it = s.iterator();

Iterator it1 = s1.iterator();

if (s.size() == s1.size())

{

while (it.hasNext() && it1.hasNext())

{

if (!it.next().equals(it1.next()))

{

return false;

}

}

}

else

{

return false;

}

return true;

}

}

class Tester

{

public static void main(String[] args)

{

Stack<Object> s = new Stack<>();

Stack<Object> s1 = new Stack<>();

Stack<Object> s2 = new Stack<>();

s.push(1);

s.push(2);

s.push(3);

s.push("bannanas");

s.push(2.05);

s.push('c');

s1.push(1);

s1.push(2);

s1.push(3);

s1.push("bannanas");

s1.push(2.05);

s1.push('c');

System.out.println(AreEqualStacks.areEqual(s, s1));

s2.push(1);

s2.push(2);

s2.push(3);

s2.push("bannanas");

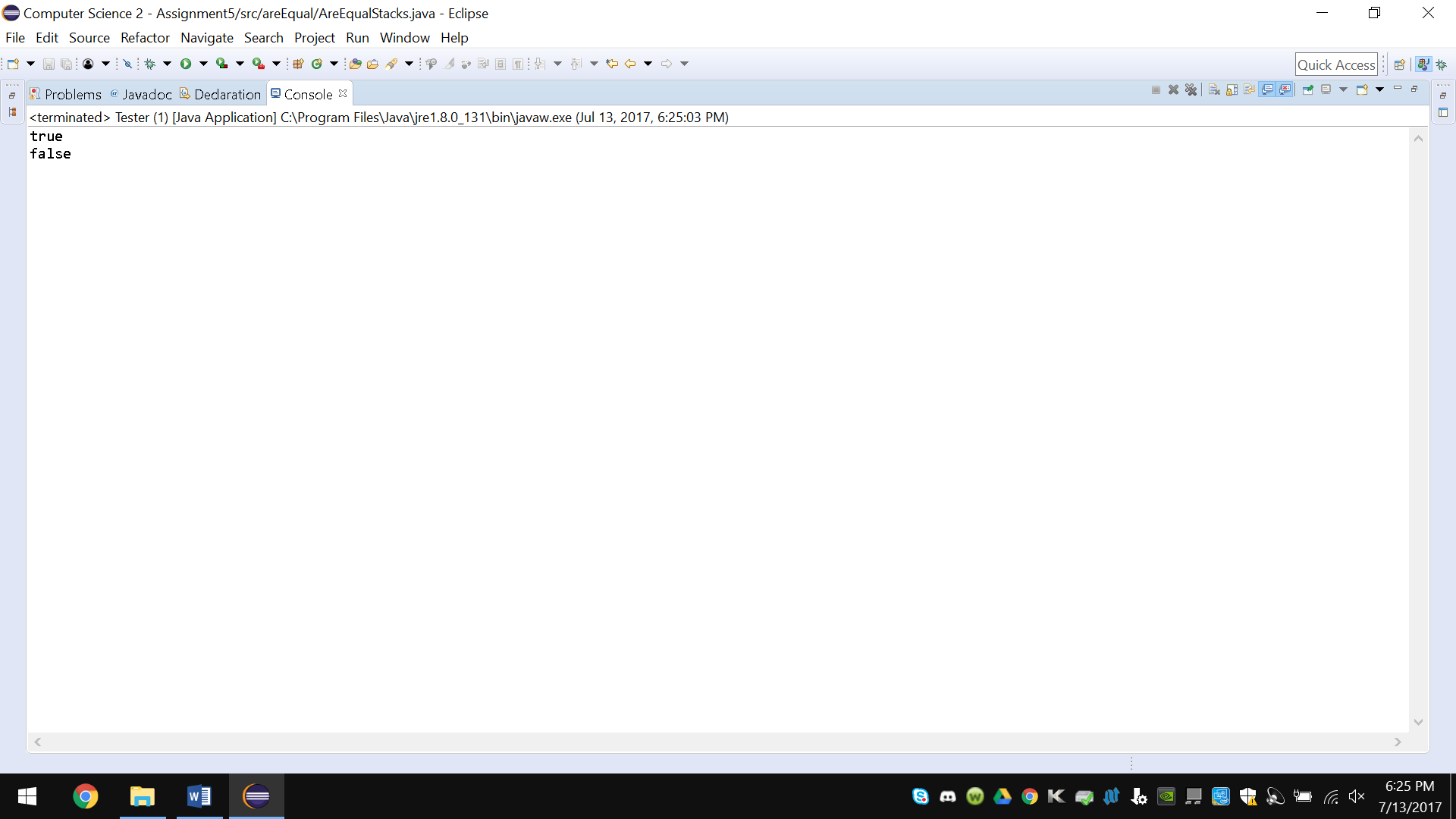
s2.push(2.05);

s2.push('a');

System.out.println(AreEqualStacks.areEqual(s, s2));

}

}



Problem 4:

Write a method duplicateStack that returns a new stack containing the same elements and in the same order as the stack specified in the parameter. The method should create a new stack and fill it with the same data elements as the given stack. (You do not need to duplicate the contents of the elements.) Before the method finishes, it must restore the contents of the original stack to its original state (same contents in the same order). Besides the new stack that the method returns, the only additional data structure that it can use is a single queue. The method may also use O(1) additional space.

package duplicateStack;

import java.util.Iterator;

import java.util.Stack;

public class DuplicateStacks

{

public static Stack<Object> duplicate(Stack<Object> s)

{

Stack<Object> returnStack = new Stack<>();

Iterator<Object> it = s.iterator();

returnStack.addAll(s);

return returnStack;

}

}

class Tester

{

public static void main(String[] args)

{

Stack<Object> s = new Stack<>();

s.push(1);

s.push(2.5);

s.push("bananas");

s.push(456786545);

s.push('c');

System.out.println(DuplicateStacks.duplicate(s) + "\n\n" + s);

}

}

