# COMP 401 Final

## Thursday, Dec 10, 2015, 12:00pm

## Instructions

1. Please spread out and try and sit in alternate seats.
2. This is a closed book exam. You will not be penalized for errors in Java syntax.
3. Write on the exam itself.
4. There are:

* 8 numbered pages including this one, and any marked blank pages.
* 3 questions.
* Point values appear in brackets next to each question.

1. You should take one minute per point of credit.
2. You are not required to comment any code you write, but may get partial credit if you write appropriate comments but incorrect code.
3. If you need to make any assumptions to clarify a problem, *write your assumptions down.* Only reasonable assumptions get full credit.
4. Please inform the proctor of anything in the exam that you think is a mistake.
5. Your code will be evaluated not only for correctness, but also for time and space efficiency and *style*.
6. You cannot use any Java capabilities not covered in class.
7. Answer in the allocated space, *do not mark the question as that will interfere with Gradescope*.
8. Write on a blank page if there is not enough space to solve a problem.
9. If you do not understand some English word, do not hesitate to ask the proctor. Naturally, you are expected to know the computer science terms defined in class.

Name (in Capitals) and Onyen (In Capitals)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Pledge: I have neither given nor received unauthorized aid on this exam.

(signed)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

For survey purposes, please indicate the time at which you turned in the exam.

\_\_\_\_\_\_\_\_\_\_\_

Please do not write below

1. \_\_\_\_\_/40 2. \_\_\_\_\_/34 3. \_\_\_\_\_/56

Total: \_\_\_\_\_/ 130

1. [40 pts.] MVC (Understanding Code)

**public** **interface** IntStack **extends** KeyListener{

**public** **boolean** isEmpty();

**public** **boolean** isFull();

**public** **int** getTop() ;

**public** **void** push (**int** element) ;

**public** **void** pop() ;

**public** **void** displayStack();

}

**public** **class** AnIntStack **implements** IntStack {

**final** **static** **int** ***MAX\_SIZE*** = 2;

**int**[] contents = **new** **int**[***MAX\_SIZE***];

**int** size = 0;

**public** **boolean** isEmpty() {

**return** size == 0;

}

**public** **int** getTop() {

**return** contents[size-1];

}

**public** **boolean** isFull() {

**return** size == ***MAX\_SIZE***;

}

**public** **void** push (**int** element) {

contents[size] = element;

size++;

displayStack();

}

**public** **void** pop() {

size--;

displayStack();

}

**public** **void** keyTyped(KeyEvent e) {

push (Integer.*parseInt*("" + e.getKeyChar()));

}

**public** **void** keyPressed(KeyEvent e) {}

**public** **void** keyReleased(KeyEvent e) {}

**public** **void** displayStack() {

**if** (!isEmpty())

System.***out***.println("Top:" + getTop());

System.***out***.println("IsFull:" + isFull());

}

}

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

IntStack anIntStack = **new** AnIntStack();

anIntStack.push(1);

anIntStack.push(2);

anIntStack.pop();

JFrame aFrame = **new** JFrame();

aFrame.addKeyListener(anIntStack);

aFrame.setVisible(**true**);

}

1. [3 pts] Name a variable in the code shown that holds an observable object.
2. [3 pts] Name a variable in the code shown that holds an observer object.
3. [20 pts] Assume the user executes the main program below and types the character ‘3’ and ‘4’ in the displayed frame. Give each line of output produced by the program. If the execution causes an exception, simply describe the nature of the exception if you cannot remember its name. *Be sure to show the output before any character is typed.*
4. [14 pts] Argue why the class AnIntStack is not a model in the MVC framework by identifying (i) features of a model it is missing, and (i) features it has that should not be in a model. In other words, describe the kind of code that is missing in the stack and the kind of code it has that should be in the view and/or controller. Be as specific as you can.
5. [34 pts.] Threads and Command Objects

The code in this question builds on the code in question 1.

**public** **class** ASynchronizedStack **extends** AnIntStack {

**public** **void** displayStack() { }; // print nothing

**public** **synchronized** **void** push(**int** element) {

**if** (isFull()) {

**try** {

wait();

} **catch** (InterruptedException e) {e.printStackTrace();}

}

**super**.push(element);

notify();

}

**public** **synchronized** **void** pop() {

**if** (isEmpty()) {

**try** {

wait();

} **catch** (InterruptedException e) {e.printStackTrace();}

}

**super**.pop();

}

}

**public** **class** ADoublePopRunnable **implements** Runnable {

IntStack intStack;

**public** ADoublePopRunnable(IntStack anIntStack) {

intStack = anIntStack;

}

**public** **void** run() {

intStack.pop();

System.***out***.println("First pop done");

intStack.pop();

System.***out***.println("Second pop done");

}

}

**public** **class** ADoublePushRunnable **implements** Runnable {

IntStack intStack;

**int** element;

**public** ADoublePushRunnable(IntStack anIntStack, **int** anElement) {

intStack = anIntStack;

element = anElement;

}

**public** **void** run() {

intStack.push(element);

System.***out***.println("First push done");

intStack.push(element);

System.***out***.println("Second push done");

}

}

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

IntStack aSynchronizedStack = **new** ASynchronizedStack();

Runnable aDoublePopRunable =

**new** ADoublePopRunnable(aSynchronizedStack);

Runnable aDoublePushRunable =

**new** ADoublePushRunnable(aSynchronizedStack, 1);

Thread aDoublePopThread = **new** Thread(aDoublePopRunable);

Thread aDoublePushThread = **new** Thread(aDoublePushRunable);

aDoublePopThread.start();

aDoublePushThread.start();

}

1. [2 pts] Name a variable in the code shown that holds a Thread object.
2. [3 pts] Name a variable in the code shown that holds a command object.
3. [20 pts] A threaded program can produce many possible outputs. Give one possible output of the main method above, identifying exception messages if they are printed. Look carefully at each overridden method in the subclass presented here, as the problem has several subtleties. You can assume in both c) and d) that InterruptedException will not be thrown.
4. [9 pts] Give another plausible output that is different from the one above, identifying any exception messages that are printed.
5. [56 pts] Recursion & Exceptions (Writing Code)

The code in this question builds on the code in question 1 – specifically the interface, IntStack. The goal here is to parse a sequence of ints that are collected in an IntStack, and determine if the sequence is a certain kind of palindrome. The only legal ints in a recognized palindrome are 0, 1 and 2, though the stack can have any int. Moreover, 2 can appear only once in the middle of the palindrome; and all other digits must be 0s and 1s. Some legal sequences:

2, 0 2 0, 1 2 1, 0 1 2 1 0, 0 0 1 2 1 0 0

Some illegal sequences

Empty sequence, 1 3 2, 0 0, 0 2, 0 1 2 0 1

Your job is to write a method that detects if the sequence of elements stored in a stack represents a palindrome, as defined above, and returns the length of the palindrome. If the stack contains an illegal palindrome, then the method should throw an exception, called IllegalPalindrome, with a message indicating the kind of error encountered. You cannot store the tokens in the stack in any data structure such as a string/array. Thus, the only data types you can use are boolan, int, IntStack, and the given exception type. You cannot create another stack. You are using the interface IntStack, not the class AnIntStack. You can assume that the implementation of IntStack passed to you already has the elements in it and behaves like AnIntStack except that it can have an arbitrary number of elements.

The following grammar describes the set of legal sequences:

<Palindrome> 🡪 2 | 0<Palindrome>0 | 1<Palindrome>1

The first rule says that a sequence with just 2 is a legal palindrome. The other two rules are recursive. They say that *if we have already detected a palindrome of size N, then we can detect a new palindrome of size N+ 2 by checking if the elements before and after the matched palindrome of size N are both 0 (1).* Use these rules to write a recursive descent parser for this problem.

The exception class is declared below:

**public** **class** IllegalPalindromeException **extends** Exception {

**public** IllegalPalindromeException (String aMessage) {

**super**(aMessage);

}

}

More specifically, on the next page, write a method that takes an IntStack as an argument and either throws one or more exceptions, or returns the length of the matched palindrome. If an exception is thrown, an appropriate message should be associated with the exception. You should terminate the code as soon as an error is detected. You are free to write a single method or a set of methods, which can be static or not. You will be graded on the kind of messages you associate with the exceptions. Assume that the implementation of IntStack passed as an argument to the method can hold *an arbitrary number of elements*. The key is to use recursion here to match corresponding elements, so look at the grammar carefully. You will get partial credit based *on the legal cases you handle correctly, the illegal cases you detect, and the specificity of the exception error messages. So do handle some cases even if you cannot handle all.* You will get some points for getting the method header completely correct. You should be able to detect *the first three legal examples* and *the first four illegal examples* given above, even if you cannot identify the recursive step. Once you do these examples, you may be able to identify the recursive step. If you give a partial solution, *please document the kind of cases you can and cannot handle.* It is better to write simple code that handle some cases correctly than write complex code that handles no cases or fewer cases than the simple code. You should probably work on a scratch paper before committing it to the exam. Write legibly, you do not get credit if we cannot read what you write.