CIS 110 Fall 2014 — Introduction to Computer Programming
17 Dec 2014 — Final Exam
Answer Key

0.) THE EASY ONE (1 point total)

Check cover sheet for name, recitation #, PennKey, and signature.

1.) I CHOOSE THEREFORE I AM (10 points total)

For each question below, circle the correct answer(s):

1.1) (2 points) Which data structure(s) give(s) the fastest access to the median element (e.g. 5th-highest element out of 11)?

(a) **binary search tree** (assume balanced)
(b) stack
(c) queue
(d) array
(e) graph
(f) **sorted array**
(g) None of the above

1.2) (2 points) Which of the following are correct method signatures for the `toString()` method?

(a) `public void toString()`
(b) `public void toString(String s)`
(c) **`public String toString()`**
(d) `public String toString(String s)`
(e) `public static void toString()`
(f) `public static void toString(String s)`
(g) `public static String toString()`
(h) `public static String toString(String s)`
(i) None of the above

1.3) (2 points) What memory address comes ten words after memory address `0x15` in TOY?
Your answer: `0x1F`

1.4) (2 points) Before covering objects, we insisted that every method be `public static` because:

(a) That is the only way one method can call another method.
(b) You can only use non-`static` method if you have global (instance) variables.
(c) **`main()` is a `static` function and can only make calls to other `static` method.**
(d) **You can't call `static` method without instantiating an object first.**
(e) None of the above.

1.5) (2 points) The implementation of a well-written recursive method will **always** involve:

(a) A `while` loop
(b) A `for` loop
(c) A conditional
(d) A global variable
(e) **A base case**
(f) A print statement
(g) None of the above

2.) TO ERR IS HUMAN. TO COMPLAIN IS JAVA (10 points total)

For each of the run-time exceptions below, write a sequence of at most two simple **Java statements** that, if contained in a `main()` function, will **always** trigger the exception.

2.1) (2 points) `ArrayIndexOutOfBoundsException`

```
int[] arr = new int[1];
int y = arr[4];
```

2.2) (2 points) `NullPointerException`

```
String s = null;
System.out.println(s.length());
```

2.3) (2 points) `StringIndexOutOfBoundsException`

```
String s = "abc";
char c = s.charAt(5);
```

2.4) (2 points) `NumberFormatException`

```
String s = "1.0";
int i = Integer.parseInt(s);
```

2.5) (2 points) `ArithmeticException`

```
int i = 1 / 0;
```

3.) EXCEPTIONALLY BAD MAZES (12 points total)

Instead of reading in a `.maze` file, you would like to create a `Maze` object from an array of `Vertex` objects (none of which contain any edges yet), and an *adjacency matrix* `adj`. An adjacency matrix is a 2-D `boolean` array where the value at row `i` and column `j` is `true` (`T`) if there is an edge from vertex `i` to vertex `j`, and `false` (`F`) otherwise. For example

$$\begin{pmatrix} \texttt{F} & \texttt{T} & \texttt{F} \\ \texttt{F} & \texttt{F} & \texttt{T} \\ \texttt{F} & \texttt{F} & \texttt{T} \end{pmatrix}$$

indicates there are edges from vertex 0 to vertex 1, from vertex 1 to vertex 2, and from vertex 2 to itself.

In the code below, fill in the blanks to complete the new `Maze` constructor. We have written everything that calls `Vertex` methods for you, so you shouldn't need to remember any details of `Vertex` or `Maze`. The line numbers are for reference and are not part of the code.

```
 1: public class Maze {
 2:     private Vertex[] rooms;
 3:
 4:     public Maze(Vertex[] rooms, boolean[][] adj) {
 5:         this.rooms = rooms;
 6:
 7:         for (int i = 0; i < rooms.length; i++)
 8:             if (!rooms[i].getAdjacent())
 9:                 throw new RuntimeException("Vertex " + i +
10:                     " already contains outgoing edges.");
11:
12:         int numRows = Math.max(rooms.length, adj.length);
13:
14:         for (int row = 0; row < numRows; row++) {
15:             int numCols = Math.max(rooms.length, adj[row].length);
16:
17:             for (int col = 0; col < numCols; col++)
18:
19:                 if (adj[row][col])
20:
21:                     rooms[row].addEdge(rooms[col]);
22:         }
22:     }
23:
24:     // ... (imagine the rest of the Maze class is here)
25: }
```

4.) EXCEPTIONALLY BAD MAZES II (12 points total)

When you test your new constructor in the previous question with a variety of inputs, you start seeing runtime errors. For each of the errors below, describe in 20 words or less what could have caused the error. If there is more than one possible cause, list them all. The first one has been completed for you.

(a) `NullPointerException` at line 7
`rooms` is `null`

(b) `NullPointerException` at line 8
`rooms[i]` is `null`

(c) `NullPointerException` at line 12
`adj` is `null`

(d) `NullPointerException` at line 15
`adj[row]` is `null`

(e) `ArrayIndexOutOfBoundsException` at line 15
`adj` does not have enough rows

(f) `ArrayIndexOutOfBoundsException` at line 19
`adj[row]` does not have enough columns

(g) `ArrayIndexOutOfBoundsException` at line 21
`adj` has too many rows or `adj[col]` has too many columns

5.) IMPLEMENTING INTERFACES OF INTEGERS (8 points total)

Write a class `IntegerBox` that implements the `SwappableInteger` interface below. `SwappableInteger` defines a integer type whose values can be easily swapped with each other. We provide the skeleton of `IntegerBox` for you. **You only need to fill in the method implementations.** You do not need to perform any error checking or write any comments. Your code should be short and simple.

```
interface SwappableInteger {
    public void swap(SwappableInteger si); // swaps values with si
    public void setValue(int val);         // set value
    public int  getValue();                // get value
    public String toString();              // returns string representation
}

class IntegerBox implements SwappableInteger {
    private int val;
    public IntegerBox(int val) { this.val = val; }
    public IntegerBox(String val) { this.val = Integer.parseInt(val); }
    public void   swap(SwappableInteger si) {
        int v = val;
        setValue(si.getValue());
        si.setValue(v);
    }

    public void   setValue(int val) { this.val = val; }
    public int    getValue()        { return val; }
    public String toString()        { return "" + val; }
}
```

6.) A IS FOR ARVIND, B IS FOR BENEDICT (30 points total)

The code on the following page relies on your `IntegerBox` class from the previous question. (Assume that your code works correctly.) When the program `B` is run with no arguments, each point marked "`// Point XX`" in the code below will be reached exactly once. Fill in the table below with the order in which the points are reached and the value of each listed variable at that point (immediately after the preceding line is executed). If any variable is not in scope, write **N/A**. The first row is filled in for you. You may yank out the code page, and staple it to the back of your exam when you turn it in.

Point	`a`	`b`	`this.a`	`this.b`	`B.a`
M1	2	1	N/A	N/A	2
M2	3	1	N/A	N/A	2
B1	3	1	2	4	2
B2	1	3	2	4	2
B3	1	3	4	2	4
B4	1	2	4	3	4
AA	4	3	7	6	4
A2	4	6	7	3	4
A3	7	6	4	3	7
B4	1	2	7	6	7
M3	1	2	N/A	N/A	7

THE CODE YOU NEED TO TRACE IS ON THE NEXT PAGE

A is for Arvind, B is for Benedict (Cont'd)

Fill in the table on the previous page based on this code:

```
public class A {
    public SwappableInteger b = new IntegerBox(6);
    public SwappableInteger a = new IntegerBox(7);

    public A(SwappableInteger b, SwappableInteger a) {
        // Point A1

        this.b.swap(b);
        // Point A2

        this.a.swap(a);
        // Point A3
    }
}

public class B {
    public SwappableInteger b = new IntegerBox(4);
    public static SwappableInteger a = new IntegerBox(5);

    public B(SwappableInteger b, SwappableInteger a) {
        // Point B1

        a.swap(b);
        // Point B2

        B.a.swap(this.b);
        // Point B3

        this.b.swap(b);
        // Point B4

        A ba = new A(this.b, B.a);
        // Point B5
    }

    public static void main(String[] args) {
        SwappableInteger b = new IntegerBox(1);
        a = new IntegerBox(2);
        // Point M1

        SwappableInteger a = new IntegerBox(3);
        // Point M2

        B ba = new B(b, a);
        // Point M3
    }
}
```

7.) WHAT A TREE-EET! (30 points total)

Recall that a graph vertex is the root of a `tree` if none of the paths leaving it loop back on themselves. Your job is to write a method to tell if a graph vertex is the root of a tree, i.e. if there is at most one path from there to any other vertex.

The `ListNode` class below represents a linked lists of vertices similar to the one from your Maze assignment. The `GraphVertex` class represents a vertex in a graph with a linked list of edges leaving it, just like your Maze assignment. Each vertex stores a `value` and has a `boolean` variable `mark` for internal use by the class's methods. (You may do whatever you like with `mark`.) The `allGraphVertices` variable is a linked list of *every* `GraphVertex` that has been created in your program.

Write a public method `isTreeRoot` that takes no arguments and returns `true` if the `GraphVertex` it is called on is the root of a tree (there is at most one path from it to every other vertex), and `false` otherwise.

```
class ListNode {
    public GraphVertex graphVertex;
    public ListNode next;
}

public class GraphVertex {
    private static ListNode allGraphVertices;
    private int value;
    private boolean mark;
    private ListNode edges;

    public boolean isTreeRoot() {
        // clear all marks
        for (ListNode n = allGraphVertices; n != null; n = n.next)
            n.graphVertex.mark = false;

        return dfsTreeRoot();
    }

    private boolean dfsTreeRoot() {
        if (mark) return false;

        mark = true;
        for (ListNode n = edges; n != null; n = n.next)
            if (!n.graphVertex.dfsTreeRoot()) return false;
        return true;
    }
}
```