CS 12 — Fall 2007 — Mid-term exam **solutions**

5 PTS In the *Game of Life*, we note that some universes become *static*—that is, no cell changes state from one generation to the next. Such a static universe is a special case of a *repeating universe*, where every k generations, the states of the cells repeat. Thus, a static universe is really a repeating universe where $k = 1$.

Question: Is it possible for a universe *not* to repeat? Explain your answer.

Answer: A universe cannot repeat. There are two essential reasons for this lack of repetition:

(a) **Fixed rules:** The rules never change. If a particular configuration of live cells appears at both generation k and generation k', then the same configuration must appear at $k+1$ and $k'+1$.

(b) **Fixed number of cells and states:** For an mn universe, there are mn cells, each of which is either alive or dead. There are therefore 2^{mn} possible states for the universe. Thus, for some generation $g \leq 2^{mn}$, there must be another generation $g' < g$ that has precisely the same state.

Discussion: Most people answered this question correctly, but the reasoning was often unclear or incorrect. I accepted any statement about a fixed or finite number of cells, although it is most correct to say that the number of possible states is *bounded*. (Note that something can be finite without being bounded.) Mentioning the fixed rules was also a good thing to do, but I did not require it.

10 PTS If the method below were called with an argument of 3, what output would appear?

```
public static int factorial (int n) {

    System.out.println("In: " + n);
    if ((n == 0) || (n == 1)) {
        System.out.println("BC out: 1");
        return 1;
    } else {
        System.out.println("Recur...");
        int m = factorial(n - 1);
        int r = n * m;
        System.out.println("Out: " + r);
        return n * factorial(n - 1);
    }

}
```

Answer: The output would be:

```
In: 3
Recur...
In: 3
Recur...
In: 1
BC out: 1
Out: 2
In: 1
BC out: 1
Out: 6
In: 2
Recur...
In: 1
BC out: 1
Out: 2
In: 1
BC out: 1
```

Discussion: This question is not the one that I *meant* to ask. However, my error resulted in a twist that required you to read and think far more carefully. Specifically, note that although the recursion has already happened by the time the ``Out: '' message is printed, the line that follows *performs the recursion again.* Consequently, the output sequence is trickier than it normally would be.

Many of you wrote a shorter output sequence that would have been correct if the final line of the recursive case had read, more simply, ``return r;''. I took away only two points for missing that extra recursion because the form of the code—redundantly repeating the recursive step—is unusual and not what I would ever recommend you write in a real program.

25 PTS What is the output of running `java Foo`, given the code below?

```
public class Thing {

    public int _a;
    public int _b;

    public Thing () {
        _a = 0;
        _b = 0;
    }

    public String toString () {
        return ("" + _a + ' ' + _b);
    }

}

public class Foo {

    protected static Thing k;

    public static void main (String[] args) {

        double d = 2.1;
        k = new Thing();
        Thing t = new Thing();
        t._a = 5;
        t._b = -1;

        doStuff(t, d);

        System.out.println("d = " + d);
        System.out.println("t = " + t);
        System.out.println("k = " + k);

    }

    protected static void doStuff (Thing t, double d) {

        d = d * 2;
        for (double q = 1.0; q < d; q = q + 1) {

            t._a = t._a + 1;
            t._b = t._b * -1;

        }

        Thing k = new Thing();
        k._b = t._a;
        k._a = t._b;

    }

}
```

Answer: The output is:

```
d = 2.1
t = 9 -1
k = 0 0
```

Discussion: There are three key variables in this code, and what happens with each of them is critical to the question:

(a) **d:** There is both a local variable `d` within `main()` and a local parameter `d` within `doStuff()`. The former is initialized as `2.1` within `main()`. Although it is then passed into the latter, that pass is performed by making a copy. So, when `d` within `doStuff()` is doubled, that doubling is performed only on the local copy, while the `d` within `main()` is unaffected.

(b) **t:** Like `d`, there are local variables with this same name in both methods. Unlike `d`, these variables `t` are pointers. So, `t` is initialized to point to a new `Thing` within `main()`. The call from `main()` to `doStuff()` makes a copy *of the pointer*, leaving `t` in both methods *pointing to that same object.* Thus, any changes to the object that occur while `doStuff()` is running persist when control returns to `main()`.

(c) **k:** There are also two spaces named `k`—one that is a static class member, and the other that is a local variable in `doStuff()`. The latter *shadows* (that is, obscures) the former. Thus, the assignments performed through `k` at the end of `doStuff()` are being performed through a pointer (and to an object) that will disappear when the method ends, leaving the class variable still pointing to the same, unmodified `Thing`.

A few of you believed that the code, as presented, would not compile because of the shadowing declaration of `k` within `doStuff()`. While I do make mistakes in writing tests, I do compile and run any code that I present to be sure that it is correct. I may be mean, and the questions may be subtle or rely on critical details, but I would not present such a "trick" question, implying that the code produces an output when it could not compile.

25 PTS Write a *container class* named `DynamicArray` that stores pointers to `Object`. The class must support the following capabilities:

- **Store:** Store a pointer to some `Object` at index k.
- **Retrieve:** Get a pointer to the `Object` at index k. If the entry at index k has never been assigned, return `null`.
- **Length:** Return the highest index value into which a pointer has been stored.

Unlike normal arrays, when a programmer creates a new `DynamicArray`, she should *not* specify its size. Instead, the size should automatically change to handle calls to store new pointers. This new class should use regular arrays to store the pointers internally.

Answer: Here is one possible solution:

```
public class DynamicArray {

  protected Object[] _array;

  public DynamicArray () {
    _array = new Object[0];
  }

  public void store (int k, Object o) {
    if (k >= _array.length) {
      Object[] newArray = new Object[k + 1];
      for (int i = 0; i < _array.length; i++) {
        newArray[i] = _array[i];
      }
      _array = newArray;
    }
    _array[k] = o;
  }

  public void retrieve (int k) {
    return (k >= _array.length) ? null : _array[k];
  }

  public int length () {
    return _array.length - 1;
  }

} // class DynamicArray
```

Discussion: Failure to read carefully caused a great deal of trouble with this question. The goal was to write a class that allowed a caller to insert an object *at an arbitrary index*. Many of you wrote code that linearly inserted pointers to objects an sequentially increasing indices. There was also little care taken to handle requests to retrieve pointers that resided at indices that may be higher than those used for storing thus far. Missing these points led to solutions of a much simpler problem, dodging the real challenges of this question.

35 PTS Consider *Pascal's Triangle*, which begins like this:

```
     1
    1 1
   1 2 1
  1 3 3 1
 1 4 6 4 1
```

Generally, element (i, j)—that is, the j^{th} number from row i—is the sum of elements $(i-1, j-1)$ and $(i-1, j)$. In other words, a given value is the sum of the two values immediately above it. For *fringe values*—ones for which either or both of $(i-1, j-1)$ and $(i-1, j)$ do not exist (they are outside of the triangle), the value at (i, j) is 1.

Question: Write a method that constructs and returns a pointer to Pascal's Triangle as a two-dimensional array of integers. Note that the second-dimension arrays should be *right-sized*—that is, they should be exactly as long as they need to be to hold the values in that row of the triangle.

Answer: Here is one example of a method that works. Note that it must accept, as a parameter, the number of rows of the triangle to generate.

```
    public static int[][] makeTriangle (int rows) {

int[][] triangle = new int[rows][];
for (int row = 0; row < rows; row++) {

    int columns = row + 1;
    triangle[row] = new int[columns];
    for (int column = 0; column < columns; column++) {

if ((column == 0) || (column == columns - 1)) {
    triangle[row][column] = 1;
} else {
    triangle[row][column] =
triangle[row - 1][column - 1] +
triangle[row - 1][column];
}

    }

}

return triangle;

    }
```

Discussion: The biggest problem with the question was that it was the *last* question on a time-pressured exam. There were occassional mistakes in allocating a 2-D array that had the right shape (that is, one that was not rectangular). Some used more complex code than necessary to find the edges and set those values to 1. Overall, though, the answers to this question were solid, and at least on the right track.