Introduction to Computer Science II
Spring 2020
Sample mid-term exam — Solutions

1. Questions: Provide short answers (a few sentences) to each of the following questions:

   (a) What is an *abstract class*?

   (b) Why should instance variables never be declared *public*?

   (c) Consider some method `foo()` that calls another method `bar()`. Assume that `bar()` may throw a `ThisOrThatException`, which is a *checked*[1] exception type. How must `foo()` be written to address this characteristic of `bar()`?

   Answers:

   (a) An *abstract class* defines a type of uninstantiable object that include at least one *abstract method*—a method signature with no body.

   (b) With *private* data members, only the methods of that class can access those variables. Therefore, the methods have full control over what is stored in those variables, how those variables are used and manipulated, and how the information stored in those variables is represented and accessed through *public* methods. Moreover, any bugs in the use of those variables must exist only in the methods of the class, narrowing the potential for errors.

   (c) The calling method (here, `foo()`) must either catch the `ThisOrThatException` as part of a *try-catch block*, or it must declare, in its signature, that `ThisOrThatException` is automatically rethrown.

[1]That is, not a subclass of `RuntimeException`.

Discussions:

(a) Some simply defined basic inheritence, not seeming aware that *any* class may be extended.[2] Be sure that you see the distinction. Any class may be extended; an abstract class *must* be extended to become instantiable.

To that end, many people left out one critical element—either the uninstantiability, or the requirement of abstract methods. These were smaller errors.

(b) There were a couple of types of confusion that typically arose. The first was the erroneous belief that *public* data members would somehow conflict or be ambiguous when used across multiple objects of that type. Of course, that's not a problem since each object must be accessed via a specific pointer to it. The second was that *private* data members were a layer of security against user manipulation. User's interact with the program's user interface, not directly with the code and objects. These major misunderstandings aside, most addressed at least some of the critical issues described in the answer above, earning full credit.

(c) The most common error here was to list only one of the two approaches to addressing this problem. For example, many answers illustrated the use of a *try-catch block* to catch the exception that `bar()` might throw, but then did not mention the alternative possibility of having `foo()` automatically rethrow the exception.

[2] It *is* possible to make a class uninheritable, so "any" is a bit of an exaggeration.

2. QUESTION: Consider the following doubly-recursive method:

```
public static void toZero (int n) {
    if (n == 0) {
        return;
    }
    System.out.println(n);
    if (n > 0) {
        toZero(n - 1);
        toZero(-(n - 1));
    } else if (n < 0) {
        toZero(n + 1);
        toZero(-(n + 1));
    }
}
```

Write the output that this method would generate if it were called with an argument of 4.

ANSWER:

```
4
3
2
1
-1
-2
-1
1
-3
-2
-1
1
2
1
-1
```

Discussion: This question is one for which, typically, the answer is either fully correct or tragically wrong. Many not only answered this question correctly, but also drew some helpfully clear recursive call trees.

Some indicated that the calls would produce no output. However, as I mentioned at the beginning of the exam, I would not ask a "trick question" like that. Others did manage to find a middle-point between a fully correct answer and one that was wholly incorrect. In such answers, a portion of the output was present, often in the correct order, but with gaps in the sequence. I will admit that I do not see clearly how such answers were reached. If you gave one, be sure you know how this call sequence really works.

3. QUESTION: Consider *Pascal's Triangle*, shown here in a usefully lopsided form:

```
  \col |  0  1  2  3  4  5  6
row\   |
-------+--------------------
0      |  1
1      |  1  1
2      |  1  2  1
3      |  1  3  3  1
4      |  1  4  6  4  1
5      |  1  5  10 10 5  1
6      |  1  6  15 20 15 6  1
```

Specifically, we define the value of any position in the triangle at row r and column c as:

$$T(r,c) = \begin{cases} 1 & \text{if } c = 0 \text{ or } c = r \\ T(r-1,c-1) + T(r-1,c) & \text{if } 0 < c < r \end{cases}$$

That is, the values at the edges (the left-column and right-most-diagonal) are always `1`, while the interior values are the sum of the values above-and-to-the-left and immediately-above. Further consider the following method header:

```
  public static int[][] pascal (int r)
```

Write this method such that it creates a two-dimensional matrix of integers that are the first `r` rows of Pascal's Triangle. Note that each row is one element longer than the previous one, and so too should the second dimension arrays of this matrix be.

Answer:

```
private static int[][] pascal (int r) {

    int[][] p = new int[r][];
    for (int i = 0; i < r; i += 1) {

        p[i] = new int[i+1];
        for (int j = 0; j <= i; j += 1) {
            if (j == 0 || j == i) {
                p[i][j] = 1;
            } else {
                p[i][j] = p[i-1][j-1] + p[i-1][j];
            }
        }

    }

    return p;

}
```

Discussion: One aspect of this question was far and away the most problematic: the allocation of the rows of the array. A great majority of answers performed a single allocation at the beginning, using an expression such as, `new int[r][r]`, therefore creating a matrix of integers that was *square*. Yet the last sentence of the question makes clear that each row should be allocated such that each is one longer than the previous. Many people even asked me about that issue during the exam. Few actually implemented it.

Otherwise, most successfully implemented the mathematical definition in code. There were occassional off-by-one errors with the loops, or other minor gaffes. Curiously, some noticed the recursive nature of the mathematical definition given in the question, and then tried to write a recursive method to perform the calculation. Most of these attempts did not work out, since filling a single 2D array structure through a recursive method can be tricky. As practice, try it yourself, and see if you can do it correctly.

4. QUESTION: Consider the following partially-written class:

```
public class IntArray {

    private int[] _storage;

    public IntArray (int size) {
        _storage = new int[size];
    }

    public int length () {
        return _storage.length;
    }

}
```

Notice that this is a container class for `int` values. An `IntArray` must be created to have a specific length (just like regular arrays). Moreover, *indexing* into an `IntArray` is a bit different from normal arrays. Specifically, each entry may be accessed via either of **two** indices:

(a) Its *forward index*, which is that entry's position counting forwards from the beginning of the array, represented as a **non-negative** integer; or

(b) Its *backward index*, which is that entry's position counting backwards from the end of the array, represented as a **negative** integer.

In other words, for an array of length 4, the forward indices are 0, 1, 2, and 3; the backward indices to those same 4 entries are respectively -4, -3, -2, and -1. Complete this class by adding the following methods:

- A *copy constructor*.
- A *deep equality comparison* method.
- A *setter* that sets the entry at index `i` to the value `v`. Remember that `i` may be either a *foward* or *backward* index. If `i` is invalid—too large in magnitude for length—then this method should throw an `ArrayIndexOutOfBoundsException` that contains the offending `i`.
- A *getter* that returns the value at index `i`. As with the setter, forward and backward indices are allowed, and invalid indices should trigger the appropriate exception.

Answer:

```
public IntArray (IntArray other) {
    _storage = new int[other._storage.length];
    for (int i = 0; i < _storage.length; i += 1) {
        _storage[i] = other._storage[i];
    }
}

public void set (int index, int v) {
    int i = translateIndex(index);
    _storage[i] = v;
}

public int get (int index) {
    int i = translateIndex(index);
    return _storage[i];
}

private int translateIndex (int index) {
    int i = index;
    if (i < 0) {
        i = _storage.length + i;
    }
    if (i < 0) {
        throw new ArrayIndexOutOfBoundsException(index);
    }
    return i;
}

public boolean equals (IntArray other) {
    if (_storage.length != other._storage.length) {
        return false;
    }
    for (int i = 0; i < _storage.length; i += 1) {
        if (_storage[i] != other._storage[i]) {
            return false;
        }
    }
    return true;
}
```

DISCUSSION: Perhaps the greatest limitation on the answers to this question was time. Many of you simply didn't have enough to complete all of the methods in this question. That said, a number of common mistakes did occur.

First, given an acceptable negative (a.k.a., *backward*) index, most answers calculated the correct non-negative index in the underlying storage. However, given an invalid backward index (e.g., `-10` for `IntArray` of length 5), the exception that would then be thrown would not contain the original index (-10), but rather the result of translating the index to a forward one.

The copy constructor often had the strange problem of not being written as a constructor at all—it was frequently written as a regular method named something like `copy()`. Be sure that you know what a copy constructor *is*. Likewise, some were confused by the description of a *deep equality comparison* method. That was a request for an overriden `equals()` method, which exists to do exactly that kind of deep comparison for equality.

Another common mistake, particularly for the copy constructor and for `equals()`, was the passing and returning of `int[]`. The `IntArray` object *contains* an `int[]` for storage, but these interior structures should not be passed or returned. Copying does not involve merely duplicating the array itself without creating a new `IntArray` object. In a similar vein, some copied or compared arrays by comparing their pointers, rather than the contents of those arrays. The latter, of course, is what makes the comparison *deep* (where as *shallow* comparison just compares the pointers themselves).