Introduction to Computer Science I
Spring 2016
Mid-term exam — Solutions

1. Question: Provide short answers (one to three sentences) to each of the following questions:
   (a) What is the difference between *passing/returning* values and *printing/inputting* (from the keyboard) values?
   (b) When you write Java source code, *for whom* are you writing it? (That is, who is your audience?)
   (c) Why does Java have *integer* data types? Why not use floating point numbers (e.g., *float* and *double* data types) for everything numeric?

   Answer:
   (a) Values can be *passed* into and *returned* from a method that is called; these are inputs into and outputs from one method to another method. Data is *printed* to the screen or *input* from the keyboard as part of an interaction with the user—an exchange of input and output with a human sitting at the computer.
   (b) One audience is the machine, in that the compiler will translate it into a form that the computer can then carry out. The other audience is the set of humans that may read the source code to discover how you chose to solve a problem.
   (c) Floating-point data types (e.g., *float* and *double*) can suffer from rounding errors. That is, each such value can be only so precise, and so any rounded values represent inexact results. Integers are always exact, with no rounding errors.

   Discussion:
   (a) Many of the descriptions provided were vague. Most identified that printing/inputing involves the user, but the descriptions of passing/returning were often muddled. People included assertions about the role of `main()` (which has no special role here), as well as commentary about whether values were saved or usable later (for some notion of *later*).
   (b) A common error was to claim that the code was for the *user*. Typically, a program's user never sees the source code; the user interacts with the

program, and is therefore, at most, an indirect audience of the source code.

(c) Many people claimed that floating-point numbers took up more space, were slower for the computer to use, and had a more limited range than integers. These all are false assertions. The range for floating point numbers is larger, but it's combination of range and accuracy is more complex. They're not slower—hardware for performing floating-point operations is just as fast as for integer manipulation. And a `float` takes the same 32 bits as an `int`, while both `double` and `long` values take 64 bits. Besides, the question wasn't about efficiency!

2. QUESTION: What is the output of the following code?

```
int x = -5;
if (x <= -1) {
    x = -x;
}
if ((1 <= x) && (x <= 9)) {
    x = x * 2;
} else if ((10 <= x) && (x <= 19)) {
    x = x * 10;
} else {
    x = x * 1000;
}
System.out.println(x);
```

ANSWER:

```
10
```

DISCUSSION: Yes, the answer is awfully short, but it makes clear which path through the code you believed the program would follow. The first conditional statement (that tests `x <= 1`) is a standalone statement; whether that condition is `true` or `false` will not affect whether the second condition (`(1 <= x) && (x <= 9)`) is evaluation, since it *always* will be. In this example, where `x = -5`, the first condition is true and `x` is negated to become `5`.

What follows is a chain of `if-then-else` statements. That second condition is tested, and indeed, `x` is between 1 and 9. Consequently, `x` is doubled to become `10`.

Because the following conditional statement is connected to the `else` of this second condition, *none of it will be used*. That is, because `x` was between 1 and 9, then the third condition (`(10 <= x) && (x <= 19)`) is *not* tested, even though `x` is, at that point, 10. So, `x` is neither multiplied by 10 nor 1,000. That leaves the resulting, printed value, `10`.

Most people got this one, but some got confused about the chain of events described above.

3. QUESTION: What is the output when `ArrayPass` is run?

```
public class ArrayPass {
    public static void main (String[] args) {
        int[] q = new int[15];
        int i = 0;
        while (i < q.length) {
            q[i] = i * i;
            i = i + 1;
        }
        i = 5;
        moose(i, q);
        System.out.println(i);
        System.out.println(q[i]);
    }
    public static void moose (int i, int[] a) {
        i = i + 1;
        a[i] = a[i] * 2;
    }
}
```

ANSWER:

```
5
25
```

DISCUSSION: This question is all about knowing how data is passed into methods; specifically, it is important to remember that arrays are not themselves passed, but rather *pointers* to them.

First, `main()` initializes an array (through the pointer `q`) such that each position k contains the value k^2. Then, in calling `moose()`, a copy of the value `5`, from `main()`'s variable `i`, is copied into the parameter `i` for `moose()`. That is, there are two spaces, both named `i`, that are distinct and accessible only within the method that declares each. Changes to `moose()`'s `i` do not affect the value stored in `main()`'s `i`.

However, the pointer value stored in `q` is copied into the parameter `a` when the call to `moose()` occurs. Consequently, changes to values in the array to which both of those pointers lead will persist after `moose()` returns, and be "visible" to `main()` through `q`.

That said, in this example, `moose()` will double the value at index `6`. The value at index `5` is unchanged (as is `main()`'s `i` itself). Consequently, `5` and `25` are the output produced by `main()` at the end of this program.

Most mistakes began with the erroneous belief that `i` is modified by `moose()` in a way that persists outside of it, leaving `i = 6`. For those who wrote `6` as

the first line of output, the corresponding second line, showing `q[6]`, should print `72`. Specifically, it is `q[6]` that is doubled in the call to `moose()`, and that change *does* persist after that call ends.

4. QUESTION: Write a method named `printRhombus` that, when passed a size (in this example, `5`), prints the following rhombic pattern:

```
    .....
   .....
  .....
 .....
.....
```

ANSWER:

```
public static void printRhombus (int size) {
    int row = 1;
    while (row <= size) {
        int spaces = size - row;
        printSpaces(spaces);
        printDots(size);
        System.out.println();
        row = row + 1;
    }
}

public static void printSpaces (int n) {
    int c = 1;
    while (c <= n) {
        System.out.print(" ");
        c = c + 1;
    }
}

public static void printDots (int n) {
    int c = 1;
    while (c <= n) {
        System.out.print(".");
        c = c + 1;
    }
}
```

DISCUSSION: This pattern is quite similar to the ones from Lab-4 and Project-1. By figuring out how many spaces to print (before printing the same number of dots/periods) on each row, you could loop through the rows of the pattern and print the correct number of each in sequence.
I wrote my solution, above, using helper methods (`printSpaces()` and `printDots()`), largely because that is how we've structured examples and solutions in our as-

signments. However, the loops within those helper methods could have been placed directly into `printRhombus()`, in place of the calls to the helper methods themselves. Either approach was acceptable for this question.

Most mistakes of significance involved a failure to structure the loops correctly. Some put the dots-loop inside the spaces-loop (which cannot work), some forgot to have an enclosing rows-loop, and many just botched the arithmetic relating the pieces. For example, too often the math done to decrement the number of leading spaces with each row was malformed, yielding no change in the number of spaces at all.

Finally, far, far too often, the user would be prompted for a value (typically, the size). The question clearly states that `printRhombus()` takes the size as a parameter. Some wrote `main()` methods to handle the interaction with the user on that front, and while that isn't necessarily wrong, it is certainly unrelated and superfluous.

5. (30 points) Write a method that compares two arrays of `int` to determine whether their contents are identical. It must return `true` if the contents are identical, and `false` otherwise. It's signature should be:

```
public static boolean compare (int[] a, int[] b)
```

ANSWER:

```
public static boolean compare (int[] a, int[] b) {
    if (a.length != b.length) {
        return false;
    }
    int i = 0;
    while (i < a.length) {
        if (a[i] != b[i]) {
            return false;
        }
        i = i + 1;
    }
    return true;
}
```

DISCUSSION: Many people asked, during the exam, whether (a) the arrays had to be created (*no)*), and (b) whether the arrays would have the same length (*not necessarily*). Understand that this method is one of our "black boxes," in that we should assume that pointers to two arrays are being passed to the method, and that we don't know anything about how they were made nor their lengths.

Luckily, many figured out that arrays of differing lengths tautologically could not be identical. Thus, the first test it to compare the lengths and, if differing, return `false` immediately. Remember that a `return` statement, no matter where in a method it occurs, ends that method immediately.

Therefore, the loop that follows may assume equal lengths, and then compare each pair of values in the two arrays. (Some also asked whether the order of the values mattered. *Yes.* Arrays have an ordering; something in which the ordering didn't matter would be called a *set*.) Notice, in the loop, that we need only find one pair of unequal values to determine that the arrays are not identical, and thus return `false`. Moreover, notice that we cannot return `true` until *all* of the pairs of values have been tested; to return `true` upon finding just one equal pair is insufficient.