CS 11 Spring 2010 — Mid-term #1 **Solutions**

1. (15 points) [Question:] What will the following fragment of code print? (Hint: Pay close attention to the types of variables and operators!)

```
int i = 12;
double d = i;
System.out.println("First: " + d);

int j = 18;
double e = j / i;
System.out.println("Second: " + e);

double f = (double)(j / i);
System.out.println("Third: " + f);

double g = (double)j / i;
System.out.println("Fourth: " + g);
```

[Answer:]

```
First: 12.0
Second: 1.0
Third: 1.0
Fourth: 1.5
```

[Discussion:] Overall, this problem went well for most. One of the most common problems was with the output for following `Third:`, where many indicated that it would be `1.5`. However, in calculating `f`, the division `(j / i)` is performed **before** any cast occurs. Consequently. the result of that integer division (`1`) has already discarded the fractional portion of the result, and the subsequent cast to a `double` is performed on that already truncated value, yielding `1.0`.
Startlingly, many people simply miscalculated $\frac{18}{12}$, indicating that it was $1.3333\ldots$. Another error was to assume that, without any forced casts, the code would either not compile or not run. I would not resort to such trickery on an exam; more importantly, it is critical to see that the code most certainly does compile and execute, even if the results computed are not intuitive to someone who unfamiliar with Java numerical types.

2. (15 points) [Question:] Consider the Java code below and answer the questions that follow.

```
System.out.print("Enter x: ");
int x = keyboard.nextInt();
System.out.print("Enter y: ");
int y = keyboard.nextInt();

if (x >= y) {
  System.out.println("Flip " + x);
} else if (y < 8) {
  while (x < y) {
    System.out.println("Quip " + x);
    x = x + 1;
  }
}
if (x == y) {
  System.out.println("Blip " + y);
}
```

(a) What would the output be if the user entered `8` and `8`?

(b) What would the output be if the user entered `3` and `5`?

[Answer:]

(a) Inputs of `8` and `8`:

```
Enter x: 8
Enter y: 8
Flip 8
Blip 8
```

(b) Inputs of `3` and `5`:

```
Enter x: 3
Enter y: 5
Quip 3
Quip 4
Blip 5
```

[Discussion:] For part **(a)**, some failed to notice that both the `Flip` and `Blip` output would be triggered by equal `x` and `y` values. For part **(b)**, two errors were common. First, many correctly identified the `Quip` outputs, but then missed that `x` and `y` would be equal when the *while* loop inside the *else* branch completed, triggering the `Blip` output. Second, some believed that a `Flip` line would follow the two `Quip` lines, failing to recognize that, since these outputs are generated by two separate branches of an

if-then-else statement, that `Flip` and `Quip` can never both appear.

Ultimately, this question was about your ability to follow the detailed sequence of execution for a set of intermingled conditional and loop statements. If you missed part of this question, be sure to examine the code carefully and understand the source of your error.

3. (10 points) [Question:] There is at least one bug in the method below that will prevent it from compiling. First, **mark and explain the problems**. Second, **change the method so that it will compile** (without, of course, changing the semantics of what the method computes).

```
public static double foo () {

  System.out.print("Enter a value for i: ");
  int i = keyboard.nextInt();
  System.out.print("Enter a value for j: ");
  int j = keyboard.nextInt();
  if (i > j) {

    int x = 0;
    double d;
    while (x < i) {
        d = i * j / 15 + i;
        x++;
    }
  }

  return d;

}
```

[Answer:]

- **The problems:** There are two, both involving the variable `d`. First, the final statement, `return d;`, will not compile because the variable `d` is *out of scope*. Specifically, its scope begins at its declaration (`double d;`), and ends at the closing brace of the *then-branch* for the statement `if (i > j)`. Second, the compiler will object to the possibility that `d` may never be assigned a value before the `return` statement attempts to return the contents of that space.
- **The solutions:** First, move the declaration, `double d;`, outside of the *then* branch so that it preceeds the *if* statement. Second, assign `d` an initial value, ensuring that there cannot be an attempt to return an undefined value. Thus, the line should read `double d = 0.0;`.

[Discussion:] Nearly everyone recognized the problem of scope and noted it. Most solved it by moving the delcaration of `d` to before the *if* statement; a few others chose to move the *return* statement into the *then*-branch. Either was acceptable.
Some missed that, if $i \leq j$, then it was possible that `d` would never be assigned a value—something the compiler would not permit as a possibility. Most solved this problem by assigned `d` some initial value. Many attempted to create an *else*-branch, but in doing so, simply emitted some kind of message to the user. The problem with

this attempted solution is that it would do nothing to placate the compiler. The mere existence of an *else*-branch does nothing by itself to ensure that `d` is assigned some value.

There was a tendency for some to identify the absence of an *else*-branch as a fundamental problem in and of itself. However, that is not the case. The fundamental problem was, as mentioned above, the lack of a return value when $i \leq j$. Whether that situation is handled with an *else*-branch or not is irrelevant.

Finally, a great many cited type problems with the computation, which employs `int` values to produce a `double` result. Although nobody lost credit for this observation alone, it is not an error. Performing arithmetic on integers and producing a floating-point result can be intentional, and is not necessarily an error. The only way to determine whether this calculation is erroneous is by knowing the context in which it is used—something not provided here.

4. (30 points) [QUESTION:] Consider the value $k!$, where the ! symbol is the *factorial operator*, and k is a non-negative integer. The computation of this value can be expressed a number of ways:

   - $k! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (k-1) \cdot k$
   - $k! = \prod_{i=1}^{k} i$
   - $k! = \begin{cases} 1 & \text{if } k = 0 \\ k \cdot (k-1)! & \text{if } k \geq 1 \end{cases}$

   Your task is to **write two versions of a method that computes the factorial function**. Specifically, the method should be named `fact`, should accept one parameter `k`, and should return $k!$. The two versions that you must write are:

   (a) An *iterative* version: compute $k!$ using a loop.

   (b) A *recursive* version: compute $k!$ by having the method call itself.

[ANSWER:] We will take each answer in turn:

(a) **Iterative:**

```
public static long factorial (int k) {

    long total = 1;

    for (int i = 1; i <= k; i++) {
        total = total * i;
    }

    return total;

}
```

(b) **Recursive:**

```
public static long factorial (int k) {

    if (k == 0) {
        return 1;
    }

    return k * factorial(k-1);

}
```

[DISCUSSION:] This problem was not only worth the most points, but did the most

damage to many people's final scores. There were a disconcerting group of people who, by ignoring or failing to understand the instructions, attempted to obtain the value of `k` from the user by printing a prompt and reading from the keyboard. Some failed to remember how to construct a recursive solution all together. Many small errors cost no points: for example, the use of `k!` as a variable name, when the exclamation point is not allowed as part of a legal variable name.

Interestingly, many people had better solutions for the recursive method than for the iterative: shorter, clearer, and more often correct. For both problems, there was a startling tendency for needless complexity. For the iterative solution in particular, some used as many as five or six variables to compute the given factorial value, often confusing themselves and using at least one of the variables in the wrong place. Others performed a nearly laughable maze of computations that, given just a little thought, were easily simplified. I penalized such solutions only in extreme cases, where the obfuscation was severe and the simplification, in places, simple.