Introduction to Computer Science I
Spring 2019
Mid-term exam — Solutions

1. Questions: Provide short answers to each of the following questions:

   (a) What does the compiler (`javac`) do?

   (b) What is *casting*? When does *automatic casting* occur? When is *forced casting* required?

   (c) What three tasks does the `new` operator perform when creating a new array? For example:

   ```
   int[] x = new int[5];
   ```

   Answers:

   (a) The compiler translations the human-readable source code (e.g., the `.java` file) into a form that the machine can run (e.g., the `.class` file). The compiler also checks that operations and methods operate on the correct types of data.

   (b) Casting converts data of one type into another type. When casting a datum can cause no loss of information (e.g., converting an `int` to a `float`), then Java will do it automatically so that it can operate on like types. When the cast may lose information (e.g., the digits after the decimal in converting a `float` to an `int`), the cast must be forced by the programmer with an explicit annotation.

   (c) The `new` operation (a) creates space for the array, (b) initializes the values in the array to 0, and (c) returns a pointer to the array.

   Notes: None.

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

```
int x = 5;
if ((1 <= x) && (x <= 9)) {
    x = x * 2;
}
if ((10 <= x) && (x <= 19)) {
    x = x * 10;
} else if ((100 <= x) && (x <= 999)) {
    x = x * 1000;
} else {
    x = -x;
}
System.out.println(x);
```

ANSWER: `100`

NOTES: The key to this question is carefully tracking the use of `else` in connection some of the conditions but not others. The first condition (`x` between 1 and 9) is true, and thus changes `x` to be 10. That conditional statement is unconnected to the remaining conditions, which are an *if-then-else* chain. Thus, exactly one of the last three branches will be used. When the first of these (`x` between 10 and 19) is true, then `x` is updated to 100, and the condition that follows (`x` between 100 and 999) is **not** tested because it follows an `else` from the previous condition.

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

```
public class ArrayPass {
    public static void main (String[] args) {
        int[] q = new int[8];
        int i = 0;
        while (i < q.length) {
            q[i] = i * i;
            i = i + 1;
        }
        i = 5;
        moose(i, q);
        System.out.println(i);
        print(q);
    }
    public static void moose (int i, int[] a) {
        i = i + 1;
        a[i] = a[i] / 2;
    }
    public static void print (int[] a) {
        int i = 0;
        while (i < a.length) {
            System.out.println("[" + i + "] = " + a[i]);
            i = i + 1;
        }
    }
}
```

ANSWER:

```
5
[0] = 0
[1] = 1
[2] = 4
[3] = 9
[4] = 16
[5] = 25
[6] = 18
[7] = 49
```

Notes: There are two essential observations here. First, although `i` changes in `moose()`, that `i` is a local variable within that method, and is distinct from the variable `i` declared in `main()`. Thus, the change to `i` within `moose()` does not affect the value printed for `i` in `main()`, which is unchanged.

Second, and in contrast, `main()` passes its *pointer* `q` to the array, which `moose()` copies into its parameter `a`. And changes to the array made through `a` are going to persist when `moose()` completes and returns to `main()`. Therefore, when the array is printed by a call to `print()` from `main()`, the change made to entry 6 of the array will appear.

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 = 0;
        while (row < size) {
            int space = 0;
            while (space < row) {
                System.out.print(' ');
                space = space + 1;
            }
            int plus = 0;
            while (plus < size) {
                System.out.print('+');
                plus = plus + 1;
            }
            System.out.println();
            row = row + 1;
        }
    }
```

NOTES: This pattern is a straightforward variation of the patterns required in Project-1.

5. QUESTION: Write a method named *pairwiseSum* that sums each pair of elements from two arrays of `int` values. The method should store the results into a new array that it returns. The method's signature should be:

```
public static int[] pairwiseSum (int[] a, int[] b)
```

For example, here is the pairwise sum of the two arrays `a` and `b`:

```
index:  0   1   2   3   4
-------------------------
    a:  9   6   0   2  13
    b: -5   1   4   7   2
  sum:  4   7   4   9  15
```

Notice that this method should only produce a new array if both `a` and `b` have the same length. If they do not, the method should return the special value, `null`.

ANSWER:

```
    public static int[] pairwiseSum (int[] a, int[] b) {
        if (a.length != b.length) {
            return null;
        }
        int[] c = new int[a.length];
        int i = 0;
        while (i < a.length) {
            c[i] = a[i] + b[i];
            i = i + 1;
        }
        return c;
    }
```

NOTES: It was not terribly important that everyone understanding the `return null;` statement—we will soon discuss that in more depth. What mattered is that the lengths of the arrays were tested to insure that they matched.

The primary focii of this question, though, was understanding: that a new array needed to be created; that the arrays needed to be traversed (i.e., moved through, in a loop, one position at a time) in lockstep, summing the values from `a` and `b` to assign the corresponding value in `c`; and, that the pointer to the new array must be returned.