# CIS 110 — Introduction To Computer Programming

## November 21st, 2011 — Exam 2

**Answer key**

**CIS 110 Exam 2 Instructions**

- You have 50 minutes to finish this exam. Time will begin when called by a proctor and end precisely 50 minutes after that time. If you continue writing after the time is called, you will receive a zero for the exam.
- This exam is *closed-book, closed-notes, and closed-computational devices* except for a one page sheet (8.5" by 11") of double-sided notes.
- When writing code, the only abbreviations you may use are as follows:

$$\begin{aligned} \text{System.out.println} &\longrightarrow \text{S.O.PLN} \\ \text{System.out.print} &\longrightarrow \text{S.O.P} \\ \text{System.out.printf} &\longrightarrow \text{S.O.PF} \end{aligned}$$

  Otherwise all code must be written out as normal.

- Please do not separate the pages of the exam. If a page becomes loose, please make sure to write your name on it so that we don't lose it, and use the provided staplers to reattach the sheet when you turn in your exam.
- If you require extra paper, please use the backs of the exam pages or the extra sheet paper provided at the end of the exam. Clearly indicate on the question page where the graders can find the remainder of your work (e.g., "back of page" or "on extra sheet").
- If you have any questions, please raise your hand and an exam proctor will come to answer them.
- When you turn in your exam, you will be required to show ID. If you forgot to bring your ID, please talk to an exam proctor immediately.

*Good luck, have fun!*

**CIS 110 Exam 2 Cheat Sheet**

```
/** 1. For syntax, look at the format of the code presented in the questions
 *     (unless otherwise stated, it is all syntactically correct code). */

/** 2. Useful methods for String objects */
charAt(index)           // Returns the character at the given (zero-based) index.
endsWith(text)          // Returns true if the string ends with the given text.
indexOf(character)      // Returns the (zero-based) index of the given character.
length()                // Returns the length of the string.
startsWith(text)        // Returns true if the string starts with the given text.
substring(start, stop)  // Returns the characters from the start index to just
                        //   before the stop index
toLowerCase()           // Returns a new string with all lower case characters.
toUpperCase()           // Returns a new string with all upper case characters.

/** 3. Useful methods for Scanner objects */
new Scanner(src)        // Makes a new Scanner from the given source.
next()                  // Returns the next token from the Scanner.
hasNext()               // Returns true if there is a token to read.
nextLine()              // Returns the next line from the Scanner.
hasNextLine()           // Returns true if there is a line left to read.
nextX()                 // Returns the next token as a X, e.g., Int, Double.
hasNextX()              // Returns true if there is a token left and it's an X.

/** 4. Useful methods for Random objects */
new Random()            // Creates a new random object.
nextInt()               // Returns a random int.
nextInt(max)            // Returns a random int in the range 0 to max-1.
nextDouble()            // Returns a random double in the range 0.0 to 1.0.
nextBoolean()           // Returns a random boolean.

/** 5. Useful methods from the Arrays class */
Arrays.toString(arr)        // Returns a formatted String representing arr,
                            //   e.g., [1, 2, 3].
Arrays.equals(arr1, arr2)   // Returns true iff arr1 is pairwise equal to arr2.
Arrays.copyOf(arr, len)     // Returns a copy of arr up to the specified length.
                            //   truncating or padding the new array to meet it.
Arrays.fill(arr, val)       // Replaces the elements of arr with val
Arrays.deepToString(arr)        // Variants that work with
Arrays.deepEquals(arr1, arr2)   //   multidimensional arrays.
```

**Method mysteries: episode 1 — the conditional menace**

1. (10 points) Given the following method, evalute the following Java expressions to their final values.

```
public static int mystery1(int a, int b) {
    int q = 1;
    if (a + b >= 5) {
        q *= -1;
    } else if (a + b <= 5) {
        q *= 2;
    }

    if (a + b == 5) {
        q += 1;
    }
    return q;
}
```

(a) `22 < 6 || (19 > 3 && true)` $\longrightarrow$

`true`

(b) `!(!(5 > 3) && !(false == true))` $\longrightarrow$

`true`

(c) `mystery1(6, 5)` $\longrightarrow$

`-1`

(d) `mystery1(2, 3)` $\longrightarrow$

`0`

(e) `mystery1(1, 2)` $\longrightarrow$

`2`

**Method mysteries: episode 2 — indefinite attack of the clones**

2. (10 points) Given the following pair of methods, evaluate the following Java expressions to their final values.

(a)
```
public static int mystery2(int x, int y) {
    int k = 0;
    while (x != y && k < 5) {
        if (x > y) {
            x /= 2;
            y *= 2;
        } else {
            x *= 2;
            y /= 2;
        }
        k++;
    }
    return k;
}
```

i. `mystery2(64, 2)` $\longrightarrow$

   5

ii. `mystery2(5, 5)` $\longrightarrow$

   0

iii. `mystery2(2, 32)` $\longrightarrow$

   2

(b)
```
public static int mystery3(int n) {
    int s = 0;
    while (n % 10 != 0) {
        int v = n % 10;
        if (v < 5) {
            s += v * 2;
        } else {
            s += v;
        }
        n = n / 10;
    }
    return s;
}
```

i. `mystery3(15841)` $\longrightarrow$

$\boxed{25}$

ii. `mystery3(31063)` $\longrightarrow$

$\boxed{12}$

**Be assertive**

3. (15 points) For each of the labeled points in the code fragment below, identify each of the assertions in the table as being *always* true, *never* true, or *sometimes* true and sometimes false.

Assume that we never *overflow* any of the `int`s in the below code. That is, we never add so much to any one of the `int` variables that it wraps around and becomes negative when it was originally positive or vice versa.

*Note:* You may abbreviate *always* with A, *never* with N, and *sometimes* with S.

```
public static int mystery(Console in,
                          int x) {
    int y = x;
    int z = 0;
    boolean b = true;
    // POINT A
    while (b) {
        // POINT B
        y = in.nextInt();
        if (x < y) {
            z += y - x;
            // POINT C
        } else if (x == y) {
            z -= y + x;
            b = false;
            // POINT D
        }
    }
    // POINT E
    return z;
}
```

| | `b == true` | `x == y` | `z > 0` |
|---|---|---|---|
| A | Always | Always | Never |
| B | Always | Sometimes | Sometimes |
| C | Always | Never | Always |
| D | Never | Always | Sometimes |
| E | Never | Always | Sometimes |

4. (20 points) Write a method `getTotalFor` that takes a `Scanner` that is reading from a text file in a particular format, and two `String`s, a `pennkey` and a `hw` and returns the total score for the student with that `pennkey` for the given `hw` as an `int`. The file contains a series of grade entries for CIS 110 where each record represents a student's score for *a particular problem of some homework*, one record per line, in the following format:

```
<pennkey id> <hw> <name of problem> <score>
```

To calculate the total grade for a homework, you must add up all the scores for each problem record for that homework. In the event that either the `pennkey` does not appear in the file or `pennkey` does not have entries for the given `hw`, you should return `0`. You may assume that the given file contains no duplicate entries (i.e., two records containing the same homework and problem for the same person).

For example, the following is a possible text file in the correct format along with example invocations of `getTotalFor` with this text file (in a `Scanner` called `file`):

```
dunhamo    hw1 Problem1 10
bishopp    hw2 Problem1 10
broylesp   hw3 Problem1 9
dunhamo    hw1 Problem2 5
broylesp   hw3 Problem2 5
dunhamo    hw1 Problem3 3
```

| pennkey | hw | Return value |
|---|---|---|
| `"dunhamo"` | `"hw1"` | `18` $(10+5+3)$ |
| `"broylesp"` | `"hw3"` | `14` $(9+5)$ |
| `"bishopp"` | `"hw8"` | `0` |

```
public static int getTotalFor(Scanner file, String pennkey, String hw) {
    int total = 0;
    while (file.hasNextLine()) {
        Scanner line = new Scanner(file.nextLine());
        if (line.next().equals(pennkey)) {
            if (line.next().equals(hw)) {
                line.next();    // problem
                total += line.nextInt();
            }
        }
    }
    return total;
}
```

**Array whisperer**

5. (20 points) Write a method `mutate` that takes an array of `doubles` `arr`, an `int` `k`, and a `Random` object and swaps random adjacent elements `k` times in `arr`. That is, for `k` iterations, we choose a random index `i` of `arr` and swap elements with index `i+1`. If `i` is the last element of `arr` then we swap the first element of `arr` instead. Note that `mutate` does not return anything nor does it print anything out.

For example, if we called `mutate` with the array `[1.0, 3.5, 7.1, -4.2, 9.9, 5.6]`, with `k = 5` we may execute the following five swaps:

```
[1.0, 3.5, 7.1, -4.2, 9.9, 5.6]
[1.0, 7.1, 3.5, -4.2, 9.9, 5.6]     (1 <-> 2)
[5.6, 7.1, 3.5, -4.2, 9.9, 1.0]     (0 <-> 5)
[5.6, 3.5, 7.1, -4.2, 9.9, 1.0]     (1 <-> 2)
[3.5, 5.6, 7.1, -4.2, 9.9, 1.0]     (0 <-> 1)
[1.0, 5.6, 7.1, -4.2, 9.9, 3.5]     (0 <-> 5)
```

```
public static void mutate(double[] arr, int k, Random rand) {
    for (int i = 0; i < k; i++) {
        int index = rand.nextInt(arr.length);
        int neighbor = (index+1) % arr.length;
        double temp = arr[index];
        arr[index] = arr[neighbor];
        arr[neighbor] = temp;
    }
}
```

**Gonna (try to, again) fly now**

6. (25 points)

(a) Write a method `range` that takes an `int` array `arr` and returns the range of the values found in that array. The range is calculated by subtracting the minimum value of the array from the maximum value of the array (i.e., `max - min`). Here are some sample invocations of `range` and their results:

| Array | Return value |
|---|---|
| `[1][2][3]` | `2` $(3-1)$ |
| `[0]` | `0` $(0-0)$ |
| `[-3][5][2][8][-1]` | `11` $(8--3)$ |

*Hint:* The constants `Integer.MAX_VALUE` and `Integer.MIN_VALUE` may be useful here.

```
public static int range(int[] arr) {
    int min = Integer.MAX_VALUE;
    int max = Integer.MIN_VALUE;
    for (int elt : arr) {
        if (elt < min) {
            min = elt;
        }
        if (elt > max) {
            max = elt;
        }
    }
    return max - min;
}
```

(b) Using the range method from the previous section write a method `rangeAtLeastSummary` that takes an integer array `arr` and two integers `min` and `k`. For each sequence of `k` integers in `arr`, `rangeAtLeastSummary` computes whether the range of that sequence is at least `min`. That `boolean` result is then stored in a new array that `rangeAtLeastSummary` returns.

For example, the first entry of the returned array will be `true` if the first `k` elements of `arr` have a range greater-than-or-equal to `min`. You can assume that the length of `arr` is evenly divisible by `k` and `arr` has at least `k` elements. Here are some sample invocation of `rangeAtLeastSummary` and their results:

| arr | min | k | Return value |
|---|---|---|---|
| `[1, 2, 3, 4, 5, 6]` | 0 | 2 | `[true, true, true]` |
| `[5, 3, 1, 10, 2, 4]` | 5 | 3 | `[false, true]` |
| `[10, 4, 2, 6]` | 3 | 1 | `[false, false, false, false]` |
| `[-3, 5, 8, -2]` | 9 | 2 | `[false, true]` |

```
public static boolean[] rangeAtLeastSummary(int[] arr, int min, int k) {
    boolean[] summary = new boolean[arr.length / k];
    for (int i = 0; i < summary.length; i++) {
        int[] subset = new int[k];
        for (int j = 0; j < k; j++) {
            subset[j] = arr[i * k + j];
        }
        summary[i] = range(subset) >= min;
    }
    return summary;
}
```

**Scratch Paper**